



**ENHANCING CRITICAL
INFRASTRUCTURE SECURITY USING
BLUETOOTH LOW ENERGY
TRAFFIC SNIFFERS**

THESIS

José A. Gutiérrez del Arroyo, Capt, USAF
AFIT-ENG-MS-17-M-034

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-17-M-034

ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING
BLUETOOTH LOW ENERGY TRAFFIC SNIFFERS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

José A. Gutiérrez del Arroyo, B.S.E.E.

Capt, USAF

March 23, 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING
BLUETOOTH LOW ENERGY TRAFFIC SNIFFERS

THESIS

José A. Gutiérrez del Arroyo, B.S.E.E.
Capt, USAF

Committee Membership:

Maj Jason M. Bindewald, Ph.D.
Chair

Scott R. Graham, Ph.D.
Member

LTC Mason J. Rice, Ph.D.
Member

Abstract

Bluetooth Low Energy (BLE) is a wireless communications protocol used in Critical Infrastructure (CI) applications. Based on recent research trends, it is likely that the next generation of wireless sensor networks, a CI application that the Department of Defense (DoD) regularly employs in surveillance and reconnaissance missions, will include BLE as an inter-sensor communications protocol. Thus, future U.S. military missions may be directly impacted by the security of BLE. One natural way to help protect BLE sensors is to use BLE traffic sniffers to detect attacks. The primary limitation with current sniffers is that they can only capture one connection at a time, making them impractical for applications employing multiple BLE devices. This work aims to overcome that limitation to help secure the types of BLE sensor networks employed by the DoD. First, this work identifies vulnerabilities and enumerates attack vectors against a BLE wireless industrial sensor, presenting a list of security “best practices” that vendors and end-users can follow and demonstrating how users can employ BLE sniffers to detect attacks. The work then introduces BLE-Multi, an enhancement to an open-source BLE sniffer that can simultaneously and reliably capture multiple connections. Finally, the work presents and executes a methodology to evaluate BLE sniffers. Under the evaluation conditions applied, BLE-Multi achieves simultaneous capture of multiple active connections, paving the way for automated defensive tools that can be used by the DoD and security community. The contributions within are published in one journal article and one conference paper and were presented at three conferences focused on wireless security and CI protection.

Acknowledgements

I am grateful to AFIT for letting me participate in this unique and rare opportunity. To my wife, my parents, and my sister: thank you for your love, patience and support. Without you cheering me along the way, this would have not been possible. I'm also indebted to a number of colleagues and students who embarked on this journey with me and whose conversations clarified my thinking on this and other matters. Their friendship and professional collaboration was invaluable. Finally, I must acknowledge my faculty mentors, Maj Bindewald, Dr. Graham, LTC Rice, and Maj Ramsey for their support, encouragement, and guidance throughout this process.

José A. Gutiérrez del Arroyo

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
List of Figures	ix
List of Tables	xi
I. Introduction	1
1.1 Problem Statement	2
1.2 Hypothesis	3
1.3 Investigative Questions	4
1.4 Document Structure	5
II. Background	6
2.1 Bluetooth Low Energy Security	6
2.2 Other Wireless Protocols	8
2.3 Technical Overview	9
2.3.1 Attribute Protocol and Generic Attribute Profile	9
2.3.2 Security Manager	10
2.3.3 Physical and Link Layer	10
2.4 Tools	14
2.5 Summary	14
III. Securing Bluetooth Low Energy Enabled Industrial Monitors	16
3.1 Introduction	16
3.2 Attack Methodology	16
3.2.1 Selecting the Target	17
3.2.2 Performing Enumeration and Reconnaissance	17
3.2.3 Reverse Engineering the Application	18
3.2.4 Attack	19
3.3 Attack Detection	22
3.4 Recovery Methodology	23
3.5 Securing BLE Devices	25
3.5.1 Recommendations for Vendors	25
3.5.2 Recommendations for End Users	26
3.6 Conclusion	27

	Page
IV. Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections	29
4.1 Introduction	29
4.2 Problems with Current BLE Sniffers	30
4.3 Design Considerations	31
4.3.1 General Considerations	32
4.3.2 Ubertooth One Platform Considerations	33
4.4 Following a Single Connection	34
4.4.1 Synchronization	34
4.4.2 Link Layer Control	38
4.5 Following Multiple Connections	39
4.5.1 Procedure	39
4.5.2 Limitations	41
4.6 Evaluation	42
4.6.1 Experiment Design	42
4.6.2 Experiment 1: One Master to One Slave	45
4.6.3 Experiment 2: One Master to Multiple Slaves	47
4.6.4 Experiment 3: Multiple Masters to Multiple Slaves	48
4.7 Analysis	49
4.8 Conclusion	50
V. Future Work and Conclusion	52
5.1 Use Case for BLE-Multi	53
5.2 Future Work	54
5.2.1 Vulnerability Assessments	54
5.2.2 Sniffer Tuning	55
5.2.3 Capability Expansion	55
5.2.4 Mobile Masters and Slaves	56
5.2.5 Intrusion Detection System (IDS)	56
5.2.6 Intrusion Protection System (IPS)	57
5.3 Conclusion	57
Appendix A. BLE-Multi Source Code	58
1.1 firmware/bluetooth_rxtx/bluetooth_le.c	59
1.2 firmware/bluetooth_rxtx/bluetooth_le.h	63
1.3 firmware/bluetooth_rxtx/bluetooth-rxtx.c	69
1.4 firmware/bluetooth_rxtx/ubertooth_clock.h	134
1.5 firmware/common/ubertooth.h	136
1.6 host/libubertooth/src/ubertooth_control.c	143
1.7 host/libubertooth/src/ubertooth_interface.h	163
1.8 host/libubertooth/src/ubertooth.c	167

	Page
1.9 host/libubertooth/src/ubertooth.h	179
1.10 host/ubertooth-tools/src/ubertooth-btle.c	181
Bibliography	188

List of Figures

Figure	Page
1 The Bluetooth Low Energy network stack.	9
2 The BLE connection process, beginning with an advertisement.	11
3 A single BLE connection showing the effect of connection parameters.	12
4 Bluetooth Low Energy channel locations; darker lobes represent the three advertisement channels.	13
5 Onset MX1101 Temperature and Humidity data logger in operation.	17
6 Configured password “I’MS3cur3” captured by a Bluefruit LE sniffer during authentication.	18
7 Firmware upload process determined by reverse engineering the application.	20
8 Device information (a) before and (b) after the attack.	21
9 Flash memory map for nRF51822 SoC.	23
10 Connections from the MX1101 access pads to the J-LINK pins on the nRF51.	24
11 J-Link Commander writing 0xFFFFFFF to the application start address (0x014000).	24
12 Commercially-available Bluetooth Low Energy sniffers.	31
13 Relationship between anchors, connection parameters and BLE-Multi timers.	36
14 Connection Event Scheduler in BLE-Multi.	40
15 Generic experiment setup (numbered bubbles indicate active links for Experiments 1, 2 and 3).	43
16 Frame capture success of a single connection over time (95% CI for proportion).	46

Figure	Page
17 Frame capture success of two simultaneous connections with one master (95% CI for proportion).	47
18 Frame capture success of two simultaneous connections with two masters (95% CI for proportion).	49

List of Tables

Table	Page
1 Summary of BLE tools used throughout this work.	15
2 Default services exposed by the MX1101 BLE server.	18
3 Vulnerabilities found on the MX1101 (firmware ver. R57-P72.)*	20
4 Comparison of commercially-available sniffers and BLE-Multi.	31
5 Critical advertisement channel messages and their effects on BLE connections.	39
6 Critical data channel control messages and their effects on BLE connections.	39
7 Summary of control, independent and dependent variables for experiments.	44
8 Raw results for Experiment 1 (one master to one slave).	46
9 Raw results for Experiment 2 (one master to two slaves).	47
10 Raw results for Experiment 3 (two masters to two slaves, one each).	49

ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING BLUETOOTH LOW ENERGY TRAFFIC SNIFFERS

I. Introduction

Bluetooth Low Energy (BLE) [6] is a wireless communications protocol widely used in applications that benefit from its low implementation overhead and minimal energy consumption. BLE is already built into many modern laptops, tablets and smartphones, making it an ideal candidate for short-range wireless communications for Internet of Things (IoT) devices and other Critical Infrastructure (CI) applications. Due to the simplicity, reliability and widespread availability [33] of BLE, manufacturers will likely continue to employ it in future applications.

In 2014, the Secretary of the Air Force, Deborah L. James, prioritized “Balancing Today’s Readiness with Tomorrow’s Modernization” as one of her top goals for her tenure as Secretary [31]. Modernization implies the updating and upgrading of AF mission systems, as well as the CI that supports them, to stay ahead of the “threats of the future.” Unfortunately, while new technologies make CI systems more capable, more efficient and easier to manage, they can inadvertently introduce new cyber vulnerabilities. Those vulnerabilities could have a major impact on Air Force and DoD missions. In fact, Maj. Gen. James K. McLaughlin, 24th Air Force Commander, predicted in 2014 that the security of CI would be critical to the security of Air Force networks and mission systems [10].

One military application likely to be impacted by the security of BLE is Wireless Sensor Networks (WSN). The U.S. Department of Defense (DoD) regularly employs WSNs for surveillance and reconnaissance missions [50]. WSNs are built from de-

centralized sensors that collaborate to detect a target phenomenon and communicate findings to a central node [12]. Inter-sensor communications are normally achieved through mesh routing algorithms, where each sensor acts as a network hop. With the extended mesh routing capabilities available in Bluetooth v4.2 [7] and the introduction of viable multi-hop algorithms [16, 19, 30], recent research turns to BLE as an inter-sensor communications protocol for WSN applications ranging from in-hospital medical monitoring [32] to environmental monitoring [16].

Outside of WSNs, vendors already use BLE in CI applications that combine computational capabilities with physical processes, mainly in building security and automation. For example, BLE door locks provide facility managers access control mechanisms that are easy to deploy and manage [9]. BLE-enabled temperature/humidity monitors [39], airflow measurement tools [1] and compressed air pressure sensors [54] provide wireless access to key metrics that affect heating, ventilation and air conditioning (HVAC) systems. Unfortunately, several BLE locks can be opened by unauthorized users [45], and as this work shows, some HVAC-related devices are susceptible to malware and application-level attacks. Attacks to these types of systems can impact the operational success of an organization, the security of its assets and the safety of its people [27].

As the DoD continues to upgrade its CI to counter the threat of the future, the security of BLE will become essential in ensuring the success of U.S. military missions.

1.1 Problem Statement

Unfortunately, the security of BLE sensors depends on the level of protection implemented by the vendors. While the BLE specification [7] defines security mechanisms for link layer encryption, authentication and data integrity, developers often ignore these mechanisms, leaving link-layer communications completely unprotected.

In 2016 alone, published exploitation frameworks took advantage of the lack of link-layer security to perform man-in-the-middle attacks [29][48], track and locate private BLE devices [45] and deploy rogue BLE devices to steal user information [44].

Defending WSNs and other CI applications against attack is difficult for two main reasons. First, research into BLE vulnerabilities is relatively recent. Hence, the range and types of attacks that can be used against BLE have not yet been fully enumerated. Second, there is a general lack of BLE defensive tools, especially those that can detect active or imminent attacks. The natural candidate tools for attack detection are BLE traffic sniffers, but current sniffers have limitations that make them impractical for WSNs. Specifically, the Bluefruit LE [3], TI CC2540 [52] and Ubertooth One [42] can follow only one connection at a time, and the Ubertooth One fails to maintain synchronization with active connections [21], which significantly degrades its performance over time. The goal of this work is to determine if BLE sniffers can be used to enhance the security of CI applications, to include WSNs. This thesis answers the following research question:

Can BLE traffic sniffers be used to enhance CI security?

1.2 Hypothesis

The security of CI and WSNs can be enhanced with a BLE sniffer that detects attacks on multiple simultaneous connections. This is shown by (i) identifying vulnerabilities and attack vectors on CI devices, (ii) determining if those attacks can be detected by BLE traffic sniffers, (iii) expanding the capabilities of current BLE sniffers to remove their detection limitations in WSNs and (iv) developing and executing a methodology for evaluating BLE sniffers.

1.3 Investigative Questions

Respectively, the work is structured to answer investigative questions (IQ) that address each research area.

IQ1: What vulnerabilities exist on BLE devices used in CI? The first step is to expose the types of vulnerabilities and attack vectors that exist on CI BLE devices. To that end, this work executes an attack on an HVAC sensor used to monitor the temperature and humidity of an environment. The attack highlights the vulnerabilities on the device and exploits them to download malicious firmware onto the monitor. Ultimately, the types of weaknesses discovered here inform the security tool built by the rest of this work.

IQ2: Can BLE sniffers detect attacks against BLE devices? This research presents the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer. It argues that the sniffer is positioned well for detection because it can capture the attack traffic in its entirety. It also provides a method for attack recovery and lists actionable security recommendations for vendors and end-users, including using sniffers to closely monitor unsecured BLE devices.

The fact that a BLE sniffer can detect attacks supports the hypothesis that it can enhance CI security. However, to impact WSN security, current sniffers must be altered to capture more than one simultaneous connection.

IQ3: Can current BLE sniffers be altered to capture multiple connections to enhance their impact to WSNs? Unfortunately, current traffic sniffers can only follow one connection at a time, making them impractical for applications that employ more than one simultaneous connection. Additionally, some open-source sniffers lack the functionality to synchronize to active connections, causing a drastic drop in the likelihood of capture over the length of a connection.

This work extends BLE sniffer detection capabilities by introducing BLE-Multi, a

firmware solution built on the Ubertooth One [42] platform. BLE-Multi can reliably sniff multiple connections simultaneously by using a timesharing capture technique and employing a novel connection synchronization mechanism. By doing so, the work amplifies the impact that sniffers can have on WSN security.

IQ4: How can the performance of BLE sniffers be measured? As sniffers are used in future security appliances, their effectiveness will be key to the success of the systems that employ them. This work presents a technique to evaluate sniffer performance and executes a baseline evaluation of several commercial and open-source sniffers. It compares how well different BLE sniffers can capture traffic and confirms the multi-connection tracking capabilities of BLE-Multi. In the end, BLE-Multi outperforms other open-source sniffers, while capturing and synchronizing with multiple simultaneous BLE connections.

1.4 Document Structure

The bulk of this work has been presented, published or submitted for publication by the author in [20], [21] and [24]. The background sections of those publications are combined and expanded in Chapter II, which provides a summary of the state of BLE security and a technical overview of the BLE protocol. Chapter III, covering the contributions of [20] and [21], addresses IQ1 and IQ2 by presenting an attack on an industrial monitor and analysis of attack detection from the perspective of the monitor, the end-user and a BLE sniffer. Chapter III also presents a recovery mechanism and outlines recommendations for vendors and end-users on how to enhance the security of BLE. Chapter IV, covering the contents of [24], highlights the limitations of current BLE sniffers and addresses IQ3 by introducing BLE-Multi. Chapter IV also addresses IQ4 with a comparison of currently available BLE sniffers and BLE-Multi. Finally, Chapter V presents the key areas for future work in this field and concludes.

II. Background

This chapter captures the current state of BLE security and related work, motivating the need for further security research. It follows with a technical summary of the BLE protocol, as necessary to the discussion in this work.

2.1 Bluetooth Low Energy Security

The Bluetooth Special Interest Group (SIG) first introduced BLE, also known as Bluetooth Smart, within Bluetooth Core Specification v4.0 [6] as a low-energy, low-data rate wireless communications protocol. Although BLE is lumped under the Bluetooth name, it is a different protocol from Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), otherwise known as Bluetooth Classic [26]. While Bluetooth Classic can be used for high data-rate applications, like audio streaming and data sharing, BLE is designed to transmit stateful information, such as whether a lock is open/closed or the temperature of a room. Those transmissions, which require short bursts at a much lower data rate, are particularly useful for CI and IoT applications, where transmitted data often consists of stateful information.

The BLE specification [7] provides low-level security mechanisms to enhance BLE security. It defines procedures to enable device-specific authentication, data encryption and redundancy checks for data integrity at the link layer. The main procedure, called binding or pairing, outputs a set of 128-bit symmetric keys used for encryption and authentication. In 2013, Mike Ryan [46] published a security evaluation highlighting flaws with this process that left it vulnerable to an observant attacker. An attacker that can capture a pairing exchange in its entirety can crack the 128-bit symmetric encryption key and fully decrypt the BLE conversation. Since that publication, the Bluetooth SIG improved the security of the key exchange by adopting an

Elliptic Curve Diffie Hellman (ECDH) algorithm in Bluetooth v4.2 [7].

In practice, few devices implement link-layer security, instead relying upon custom higher-level security or no security at all [21]. Link-layer security mechanisms are optional because each contributes to energy consumption; increased security comes with an increased energy cost [26]. For this reason, the Bluetooth SIG places the burden on developers and vendors to make design decisions about security mechanisms. Whether developers are motivated by energy consumption, by pushing products to market quickly, or by other design decisions, the end result is that most devices lack appropriate levels of security at the link layer.

In 2016 alone, the lack of link layer security contributed to the development of multiple BLE attacks, exploitation frameworks and reconnaissance tools. Two separate teams developed man-in-the-middle frameworks that can be used against vulnerable targets [29, 48]. Both frameworks prey on devices that lack link-layer authentication and encryption. Other researchers targeted commercially-available BLE locks and were able to open 12 out of 16 from up to a quarter mile away [45], and another two teams created BLE reconnaissance tools that exfiltrate private information and track in real time the approximate location of nearby BLE devices [43, 44].

In the same year, research in BLE security and privacy violations drove the development of an active BLE privacy protection tool [14]. With BLE-Guardian, Fawaz *et al.* [14] offer a front end for authentication and a novel process to restrict connections to authenticated devices. They use jamming to hide advertisements by protected BLE devices from unauthorized users, preventing unauthorized access. While BLE-Guardian focuses on protecting the privacy of unconnected devices, it does not offer protection after a connection is initiated to those devices. Chapter IV shows vulnerabilities that can be exploited after connection initiation and which are not detected or mitigated by BLE-Guardian. With BLE-Multi, this work presents a more capable

BLE traffic sniffer to drive development of advanced security appliances, particularly those that can act on a connection after it is already initiated.

2.2 Other Wireless Protocols

Other protocols used in CI and WSN applications include those based on the IEEE 802.15.4 specification (e.g., WirelessHART and ZigBee) and on the ITU-T G.9959 recommendation (e.g., Z-Wave) [5]. Many of the security challenges found in BLE are also present in these other wireless protocols. In fact, researchers find that the attack tools and mechanisms in one wireless domain often evolve into those in used in other domains [13]. Generally, these attack frameworks are the key drivers in the development of defensive mechanisms and security appliances, especially for new and emerging protocols [17].

One of the most recent examples is Z-Wave security. Z-Wave posed a challenge to researchers because access to the proprietary protocol required signing a non-disclosure agreement [15]. Researchers began developing open-source traffic capture and packet injection tools, which enabled sophisticated attacks on Z-Wave networks. Badenhop *et al.* [5] outlined Z-Wave attacks ranging from reconnaissance to packet injection and denial-of-service. Other researchers then followed with defensive mechanisms. Fuller *et al.* [15] built a Misuse-Based Intrusion Detection System (MBIDS) that detects malicious activity captured by an open-source Z-Wave traffic sniffer, while Hall *et al.* [25] demonstrated a mechanism to perform Z-Wave device fingerprinting to protect against rogue devices.

This work seeks to advance security research in BLE by extending the capabilities of current BLE sniffers. More capable sniffers will enable future security appliances to protect devices that have been left unsecured.

2.3 Technical Overview

The key design objective for BLE is minimal energy consumption. This goal is achieved by minimizing implementation overhead and simplifying communications protocols [26]. This section highlights aspects of the BLE network stack (shown in Figure 1) necessary to the security discussion in this work.

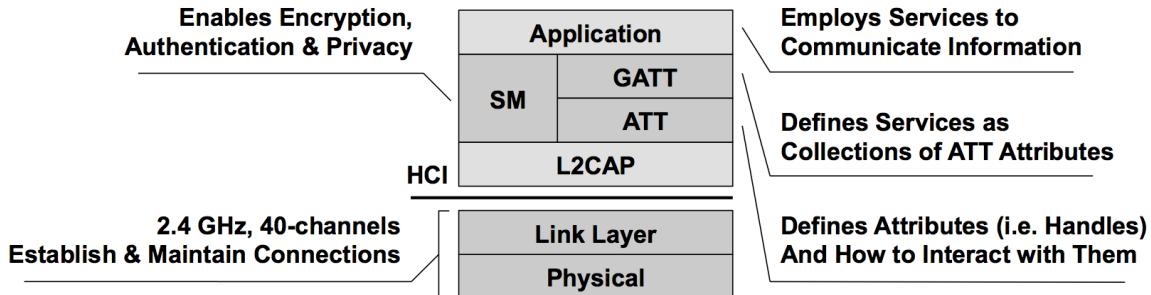


Figure 1. The Bluetooth Low Energy network stack.

GATT - Generic Attribute Profile

ATT - Attribute Protocol

L2CAP - Logical Link Control and Adaptation Protocol

SM - Security Manager

2.3.1 Attribute Protocol and Generic Attribute Profile

The Generic Attribute Profile (GATT), critical to the attack outlined in Chapter IV, defines the interactions between BLE peripherals and BLE centrals. BLE peripherals are devices that serve information (e.g., BLE locks, lightbulbs, sensors), while BLE centrals are devices that consume information (e.g., smartphones, tablets) [26]. Each peripheral contains a set of handles that can be altered by the central using the Attribute Protocol (ATT). Handles are variables that are employed to communicate state or to serve as a device control point. For example, a BLE lightbulb has a handle that holds the current state of the light (i.e., on/off). A smartphone running a

home automation application sends an ATT Read Request to read this handle and report the state back to a user. When the user decides to turn the light on or off, the application sends an ATT Write Request to the appropriate control handle, and the lightbulb changes states. The attack presented by this thesis is informed with the ATT commands that a BLE central sends to a BLE peripheral.

2.3.2 Security Manager

The Security Manager (SM) handles low-level security for BLE connections by using a device pairing and key exchange mechanism called binding [26]. While there are several pairing methods specified to fit the range of BLE applications, the output is always the same: a shared 128-bit Long Term Key (LTK) used to encrypt communications with the AES-128 block cypher encryption algorithm. They also exchange the Connection Signature Resolving Key (CSRK) and the Identity Resolving Key (IRK), which can be used for authentication and privacy, respectively. Different implementations of the SM can yield different levels of security. The devices analyzed in this work are vulnerable because they implement an SM that does not enforce encryption or authentication.

2.3.3 Physical and Link Layer

The Physical and Link Layers are responsible for establishing and maintaining BLE connections. The structure and characteristics of BLE connections are imperative to the mechanisms presented in this work to enhance BLE sniffers. BLE connections occur between two parties: (i) one master and (ii) one slave. Note that these are link-layer roles and are abstracted from the GATT/ATT roles previously discussed. In other words, while the BLE central is typically the master and the BLE peripheral is typically the slave, the opposite is also permitted by the BLE speci-

fication. Each BLE connection is comprised of a series of Connection Events (CE) with varying lengths. The master dictates when, on what frequency, and how often CEs take place. The slave generally has no control over these parameters but can choose to leave the connection at any time. Although the BLE specification allows one master to have connections to multiple slaves and one slave to have connections to multiple masters, BLE treats each link as a separate connection with its own parameters. Figure 2 provides an overview of the process, and each step is described below.

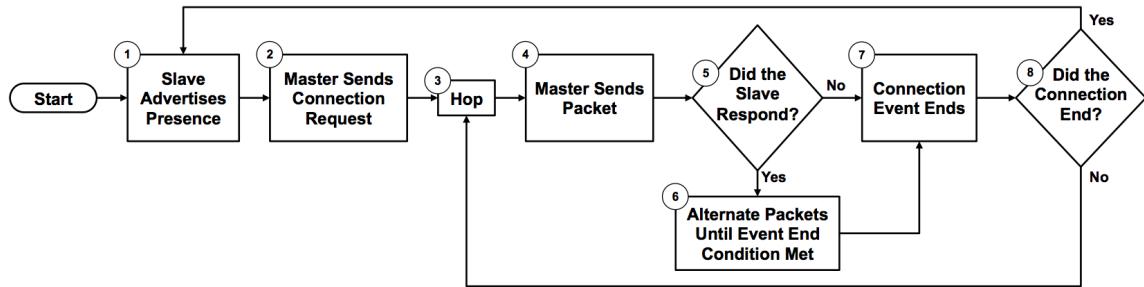


Figure 2. The BLE connection process, beginning with an advertisement.

1- Slave Advertises Presence. A slave broadcasts frames on one of three advertisement channels. Advertisements are used to announce device presence, execute device scans, and initiate connections. They also contain information about connectability and services provided by the slave. A master observes these advertisements to determine which targets are available for connection; it can only connect to a slave that advertises its presence.

2- Master Sends Connection Request. To begin a connection, the master sends a Connection Request (CR) containing critical connection parameters within $150\ \mu s$ after the end of an advertisement. The most important parameters to this discussion are the Connection Interval, Window Size, Window Offset, Hop Interval, Channel Map and master Sleep Clock Accuracy (SCA). Window Size and Window Offset determine the timing of the first CE, and the Connection Interval determines

the timing of every CE after the first. The Hop Interval and Channel Map describe the frequencies on which the master and slave communicate. Finally, the master SCA is key to the synchronization mechanism. Figure 3 shows the effect of these connection parameters.

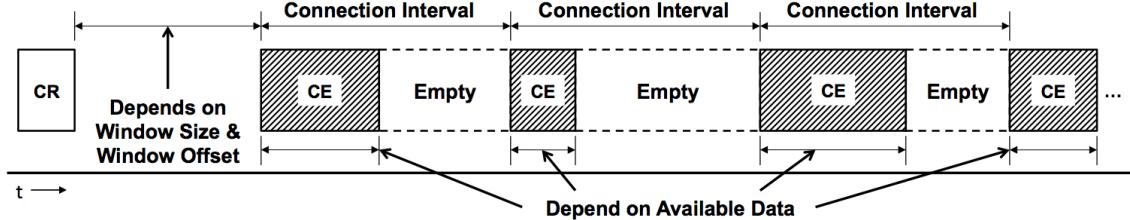


Figure 3. A single BLE connection showing the effect of connection parameters.

Note that the start time and frequency for each CE is essentially pre-determined by the connection parameters exchanged at the start of the connection. This is generally a valid assumption with two exceptions. First, the master and slave may have different expectations about where any given CE should start, depending on how far the clocks (master and slave) have drifted from each other. BLE handles this phenomenon through a synchronization mechanism that relies on the timing of the first frame in each CE. This poses challenges to the sniffer presented in this work, as third-party listeners cannot determine the directionality of captured frames. Second, the master may choose to change connection parameters at any point in the conversation. When the master makes a change, it selects a time in the future at which the change will take place, and both parties adjust the schedule of CEs accordingly. To continue following the connection, the sniffer presented in this work is required to implement those changes at the correct time.

3- Hop. BLE devices operate in the 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band. The BLE specification divides the band into 40x2-MHz channels, as illustrated in Figure 4. It reserves three channels for advertisements and uses the rest for data transmission in active connections. During a hop, both the master

and slave tune their radios to a new frequency, as defined by the Channel Map, Hop Increment and hopping algorithm. Note that any two arbitrary BLE connections will likely be talking on two different channels at any point in time. This is the primary physical limitation when expanding BLE sniffer capabilities.

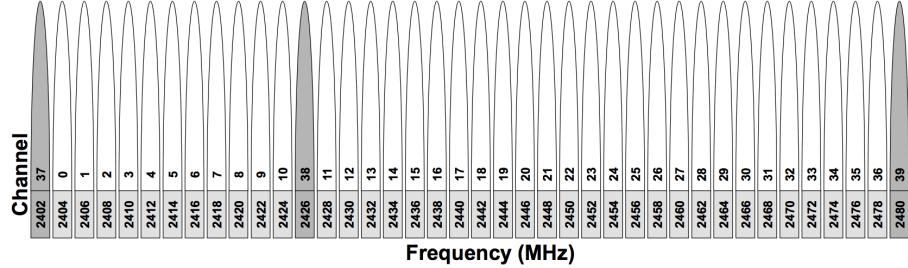


Figure 4. Bluetooth Low Energy channel locations; darker lobes represent the three advertisement channels.

4- Master Sends Packet. At the beginning of the CE, the master sends a data frame (empty or otherwise) on the appropriate frequency. This first frame is called an anchor and is critical for connection synchronization.

5- Did the Slave Respond? The slave is not required to respond to the anchor frame. It can remain inactive for a predetermined number of CEs. If the slave responds, the master and slave begin exchanging frames.

6- Alternate Packets until Connection Event End Condition Met. Once the slave responds to the anchor, the CE continues, and one or both sides can send data as required, taking turns on who transmits. Each packet contains a sequence number and a next expected sequence number, which serves as a receipt acknowledgement for the previous packet. The length of a CE is determined by how much data are available for transmission, which depends on higher layers in the BLE stack. The CE ends if both parties have no more data or acknowledgments to transmit, if neither party sends a frame within 150 μ s, or if the connection ends.

Many CEs consist of only two frames: (i) an empty anchor frame from the master and (ii) an empty acknowledgement from the slave. BLE requires frame transmission

(even of empty frames) to periodically synchronize connections and to ensure that both parties are still engaged in communications.

7- Connection Event Ends. The end of a CE does not necessarily imply the end of a connection. If the connection has not ended, the master and slave remain inactive until the master sends the anchor of the next CE.

8- Did the Connection End? If either party sends an LL_TERMINATE_IND frame, the connection ends, and the slave starts advertising its presence again.

2.4 Tools

This work employs several BLE-specific tools for traffic capture, attack and firmware recovery. As an aid to the reader, a summary of tools is provided in Table 1.

2.5 Summary

This chapter presented the state of BLE security and provided a technical overview of the BLE protocol as an aid to the reader. While there has been a recent rise in BLE security research, much of it focuses on attacks and attack frameworks and not on defense of vulnerable devices. Attacks are enabled by the fact that vendors do not implement appropriate link-level security in their applications. This same observation motivates the defensive mechanisms presented by this thesis.

Because link-level security is not robust, a BLE sniffer is a good candidate to detect attacks to these devices. However, sniffers are currently limited to one connection at a time, making them impractical for applications that require monitoring of multiple links. With BLE-Multi, this work creates a practical mechanism to monitor multiple devices. This tool can be the cornerstone of future security appliances that focus on the security of unprotected devices.

Table 1. Summary of BLE tools used throughout this work.

Tool Name	Version	Description
Ubertooth One [42]	Utilities: 2015-10-R1 Firmware: git-579f25c*	Bluetooth sniffer with open-source firmware and hardware; BLE support added in 2013 by Mike Ryan
Bluefruit LE Sniffer [3]	1.0	Commercial nRF51822-based BLE sniffer
TI CC2540 Sniffer [52]	2.18.1	Evaluation kit of the CC2540 SoC with pre-loaded sniffer firmware; used with the SmartRF Packet Sniffer software
BlueZ [8]	5.41	Linux Bluetooth stack, includes utilities to interact with BLE peripherals
Scapy [55]	2.3.2	Python library used to craft BLE messages
nRF51822 [38]	-	Multiprotocol SoC used as an all-in-one BLE deployment solution
nRF51 DK [37]	SDK: 12.0.0 Hardware: 1.2.0	Development platform for the nRF51822 SoC; includes a J-Link SWD programmer and on-board I/O
J-Link Commander [47]	5.12F	Software to interact with the J-Link SWD programmer on the nRF51 DK
Bleno [34]	1.0	Node.js module for BLE peripheral implementation

*Two bugs were discovered in Ubertooth firmware that prevented the sniffer from running indefinitely. Both were fixed and reported to the Ubertooth team [22, 23].

III. Securing Bluetooth Low Energy Enabled Industrial Monitors

3.1 Introduction

This chapter addresses investigative questions IQ1 and IQ2 of this work, which focus on identifying the types of vulnerabilities found on BLE devices and the attacks that can be used against them and determining whether BLE sniffers can detect those attacks. To tackle those subtopics, this work first executes an example attack against an industrial monitor, highlighting the vulnerabilities that enable the attack. It follows with a discussion on the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer, arguing that the sniffer is best positioned for attack detection. Finally, this chapter presents a methodology for attack recovery and provides actionable recommendations for vendors and end-users to enhance BLE security.

The contents of this chapter were originally presented by the author in the Wireless Village at DEF CON 24 [20] and in the proceedings of the Twelfth International Conference on Cyber Warfare and Security [21]. This chapter is structured as follows: Section 3.2 presents an example attack methodology, and Section 3.3 outlines the key challenges with attack detection. Sections 3.4 and 3.5 show a recovery technique and actionable mitigation steps to enhance security, and Section 3.6 concludes.

3.2 Attack Methodology

To determine the types of vulnerabilities that exist on CI devices, this section executes an attack on an HVAC sensor used to monitor the temperature and humidity of an environment. The attacker enumerates the GATT services on the target, noting which are the most important by observing ATT traffic to and from the target.

Focusing on those important services, the attacker determines when and how the BLE central and BLE peripheral communicate, determines which handles to target and performs an attack to achieve a desired effect.

3.2.1 Selecting the Target

The Onset MX1101 Temperature and Humidity Data Logger [39], shown in Figure 5, is selected as the target. The MX1101 uses the nRF51822 SoC, a common BLE chip used in many types of applications including several WSNs [28, 16]. Consequently, some of the vulnerabilities shown here apply to multiple classes of BLE devices. Under its intended use case, the MX1101, henceforth known as the “Target,” is placed in an environment to monitor for an extended period of time. A technician interacts with it using an Android or iOS device through the HOBOmobile Application [40], henceforth known as the “App.” The App configures the Target, reads and stores logged data, and updates SoC firmware.

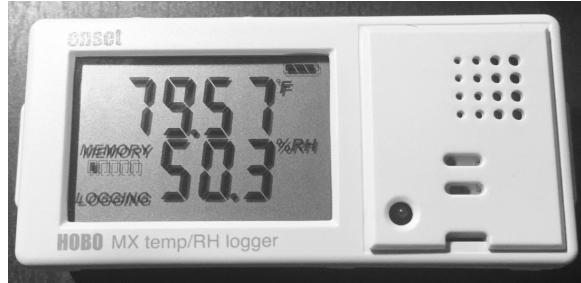


Figure 5. Onset MX1101 Temperature and Humidity data logger in operation.

3.2.2 Performing Enumeration and Reconnaissance

Using BlueZ utilities, the attacker determines that the address of the Target is D1:55:D5:00:68:C7. She uses a BLE sniffer (e.g., Bluefruit LE) to follow connections to the Target and capture frames in the conversation. Because the frames contain visible ATT Write Requests and Notifications, seen in the background of Figure 6,

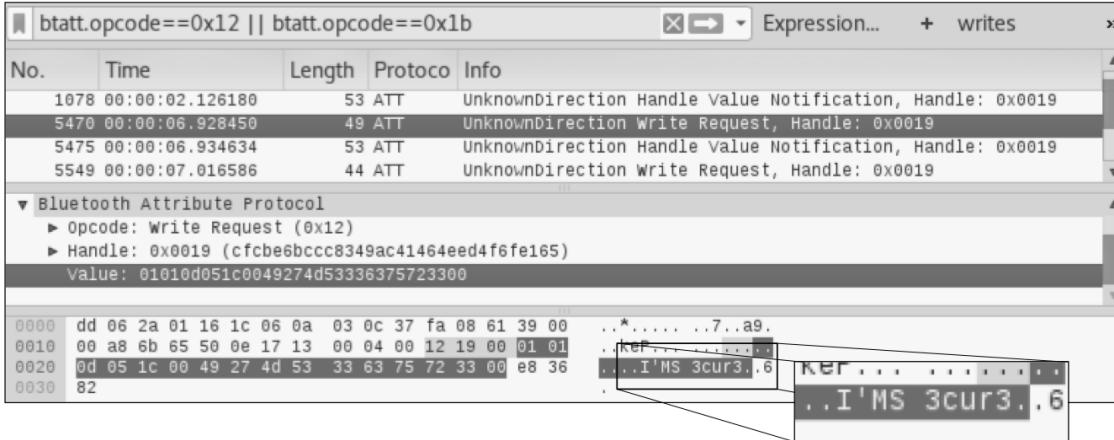


Figure 6. Configured password “I’MS3cur3” captured by a Bluefruit LE sniffer during authentication.

the SM has not enabled link encryption. In general, many devices do not use link layer encryption. This may be for “ease of use” purposes, as the devices would otherwise require pairing before communicating. Because traffic is not encrypted, the attacker extracts the Target services from the service discovery messages exchanged after connection initiation. The Target BLE server, outlined in Table 2, implements the Device Information Service defined by the Bluetooth SIG and an application-specific service defined by Onset.

Table 2. Default services exposed by the MX1101 BLE server.

Service	Handle Range	Notes
Generic Access	0x0001-0x0007	Required in BLE
Generic Attribute	0x0008-0x000b	Required in BLE
Device Information	0x000c-0x0016	Contains identification information about the device; defined by SIG
Onset Application	0x0017-0xffff	Service used by the HOBOMobile Application; defined by Onset

3.2.3 Reverse Engineering the Application

Through reconnaissance, the attacker determines that handle 0x0019 in the Onset service is a command and control (C2) handle employed by the application. To

authenticate, the App sends the configured password in cleartext to the C2 handle. Because the Target does not use encryption, a BLE sniffer captures the password when it is sent in an ATT Write Request, as shown in Figure 6.

The attacker uses a combination of open source documentation and information extracted from the App to gather details on firmware update capabilities. The Target and App user manuals claim that the Target can perform firmware updates over-the-air. Because the Target uses an nRF51822 SoC, its mechanism to update firmware is through the Device Firmware Update (DFU) Service designed by Nordic Semiconductor, the manufacturer of the SoC [35].

The Target user manual claims that the “Update Firmware” button on the App does not appear unless an update is required [39]. In other words, the DFU Service is not enabled unless the App requests it. To determine what messages are sent by the App to enable the DFU service, the attacker creates a clone of the Target using the *Bleno* Node.js module. The clone implements the same services from Table 2 and includes enough functionality to deceive the App. It reports an outdated firmware number, which triggers an update prompt from the App. When the attacker presses the “Update Firmware” button, the App sends the command 0x010108010855 to the Target to enable the DFU service.

3.2.4 Attack

After reconnaissance and reverse engineering, the attacker enumerates the vulnerabilities on the Target and select which ones to exploit to achieve an effect. Table 3 lists the vulnerabilities discovered for the Target.

The attack described below is a malicious firmware update on the Target. Figure 7 shows the general process by which the attack is executed. The following sub-sections expand on the functionality of each of the blocks.

Table 3. Vulnerabilities found on the MX1101 (firmware ver. R57-P72.)*

Vulnerability	Impact
No Default Password	The default logger configuration does not require a password for login. If left misconfigured, an attacker could connect to the device and gain full control of its operation.
Cleartext Password	The message sent to the device to perform authentication includes the logger password and is sent in plain text. This password could be easily sniffed and used to perform a legitimate login on the device to gain full control of its operation.
Unhandled Input	Once authenticated, a specific write command can trigger a system reset. The reset stops all data logging and requires reconfiguration.
Unverified Firmware	The logger does not verify the binary provided during over-the-air firmware updates. An attacker can upload arbitrary firmware to the device to affect device operations.

*Vulnerabilities were reported to Onset Computer Corporation in August 2016.

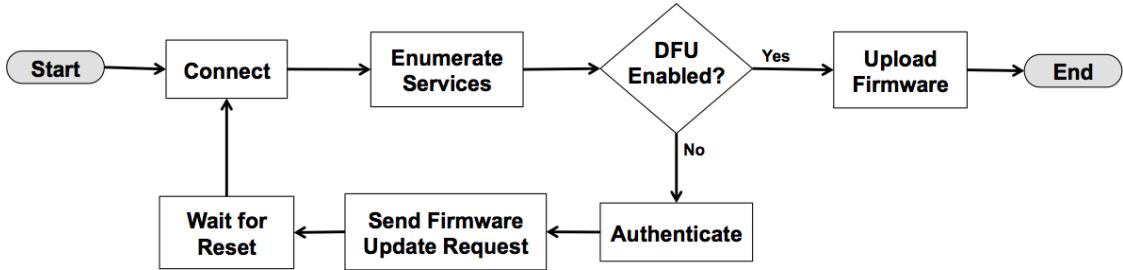


Figure 7. Firmware upload process determined by reverse engineering the application.

Connect. The attacker sends a connection request to the address captured during reconnaissance. The Target cannot authenticate the link-level connection because it does not use pairing/bonding. Furthermore, the Target does not enforce a connection timeout and allows the attacker to remain connected indefinitely. At this point, the Target is under a Denial of Service (DoS) attack because the attacker prevents legitimate users from connecting to the Target.

Enumerate Services and DFU Enabled? As in the reconnaissance stage, the attacker enumerates the services on the Target to determine whether the DFU service is enabled. If the DFU service is not enabled, the attacker authenticates with the

Target and sends the firmware update command.

Authenticate. The attacker sends an authentication request to the Target using the password captured during the reconnaissance stage. If no password is set (i.e., default operation), she sends an empty authentication request.

Send Firmware Update Request and Wait for Reset. The attacker sends the command 0x010108010855 to handle 0x0019, triggering a reboot process on the Target. After the Target resets, it enables the DFU service, which becomes visible to the attacker.

Upload Firmware. During the reconnaissance stage, the attacker determined that the Target implemented the Nordic Semiconductor DFU Service to update firmware. While the service uses a 16-bit redundancy check to guarantee the integrity of the firmware, it requires no authentication and does not verify the validity of the file. The attacker alters the original firmware file to display custom information in the Device Information Service and downloads it to the Target using a series of ATT Write Commands transmitted using a custom Scapy script. The result of the attack is shown in Figure 8.

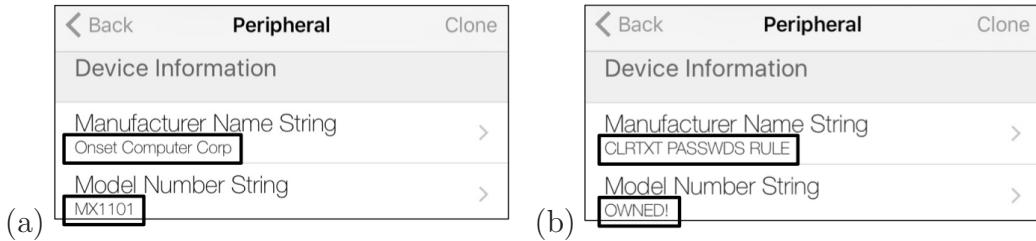


Figure 8. Device information (a) before and (b) after the attack.

While the choice of firmware here is benign, the attacker is not limited to the type of malware she can download. For example, she can create malware that presents itself as the original firmware but feeds fake data back to the App. The malware could cause the user to take action on false alarms or hide critical alarms from view,

potentially damaging the system the sensor is set to monitor. If crafted well, such malware would be difficult to detect by the App and end user. Because DFU is enabled by the App through a command to the Target, the malware can prevent any future firmware updates, significantly increasing the difficulty of device recovery.

3.3 Attack Detection

Three parties are positioned to detect an attack like the one outlined in this work: the Target, the App and a listening third party. The Target can detect an attack if it can determine that a malicious user has connected to it. Options to accomplish this include whitelisting known BLE central addresses and requiring strong link-layer authentication using shared keys. Unfortunately, BLE central addresses can be spoofed, as shown in a recent BLE security evaluation [46], and the Target cannot perform strong authentication because it does not require pairing. An end user may detect the attack if the device behaves erratically with its new firmware. However, a skilled attacker can develop malware that behaves enough like the original firmware to deceive the App and end user.

The most likely method for detection is using BLE sniffer. Since the attack is executed through a series of ATT Write Commands, a sniffer would witness and record the firmware update and could alert an attentive administrator of the change. Presence of sniffer would be difficult to detect by an attacker, as sniffers do not transmit any information. However, their use comes with its own challenges. Chapter IV highlights some the challenges of capturing traffic with sniffers and proposes novel mechanisms to overcome them.

3.4 Recovery Methodology

This section provides an overview of the method used to recover the Target after its firmware is replaced. While it is possible to completely recover the Target, the process requires opening the case, soldering onto access pads, creating a device clone using Bleno, and overwriting portions of memory using a special programmer. Not only would it be difficult for a typical user to complete this process, it would only be useful to someone who can detect that the device has been compromised.

The approach consists of re-enabling the DFU service and downloading the original firmware onto the Target. The key insight for recovery is that by exploiting the legacy DFU service, the attacker can only overwrite the application binary, leaving other sections of memory intact. If the application binary is replaced with the original firmware binary, the device returns to its original state. Unfortunately, the command to enable the DFU service, used to perform the attack in the first place, can no longer be used reliably because the new firmware may not know how to interpret it, or worse, may choose not to interpret it. Recovery requires direct interaction with the memory on the SoC.

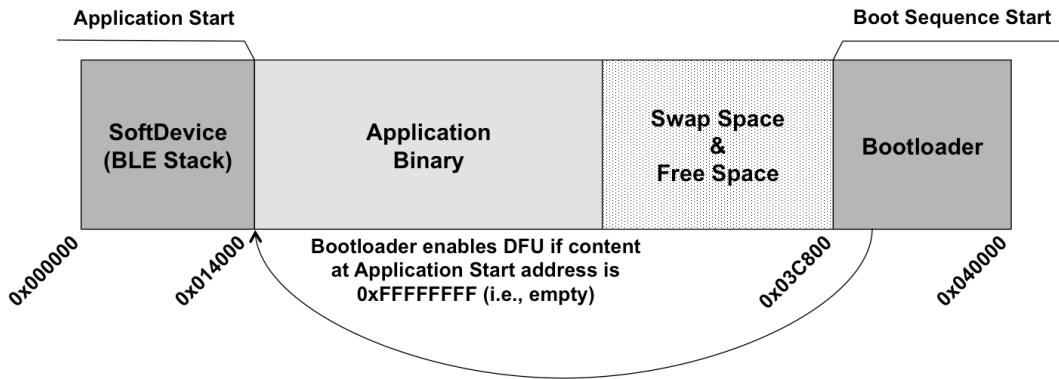


Figure 9. Flash memory map for nRF51822 SoC.

Figure 9 shows the memory layout of an nRF51822 SoC with DFU capabilities. If the content at the Application Start address is 0xFFFFFFF (i.e., empty), the

bootloader assumes no firmware is present and automatically enables the DFU service. Accordingly, the presented approach overwrites the content at 0x014000 with value 0xFFFFFFFF. The memory location is accessed directly through the Serial Wire Debug (SWD) [4] programming interface of the nRF51822 chip. On the MX1101, the programming lines (SWDIO and SWDCLK) are connected directly to a set of larger access pads, presumably for programming the device on the manufacturing line. The programmer on the nRF51 DK is connected to the Target as shown in Figure 10. Once physically connected, J-Link Commander is used to create an SWD connection with the Target, write 0xFFFFFFFF to the memory address, and perform a soft reboot, enabling the DFU service.

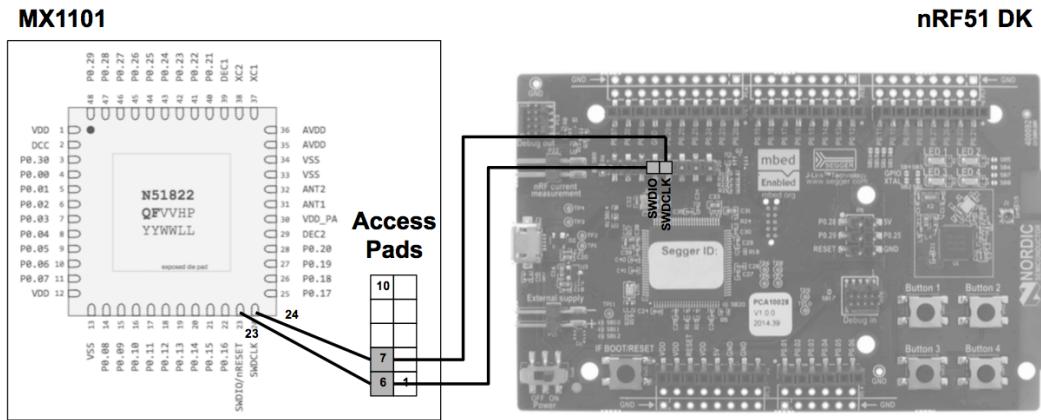


Figure 10. Connections from the MX1101 access pads to the J-LINK pins on the nRF51.

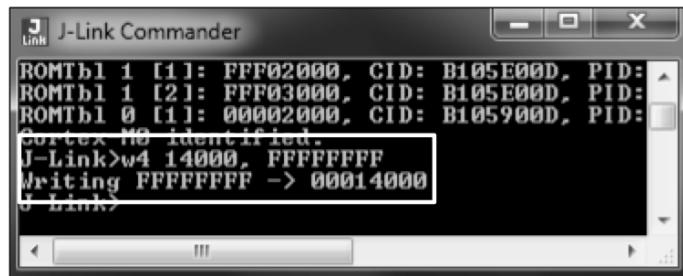


Figure 11. J-Link Commander writing 0xFFFFFFFF to the application start address (0x014000).

Once the Target enables the DFU Service, it is ready to receive new firmware. Because a normal end-user is not likely to have a copy of the original firmware, the App is employed to recover the Target. The Bleno clone is again used to trigger a firmware update from the App. After the App sends the command 0x010108010855 to the Bleno clone, it expects the device to reboot with the DFU Service enabled. Since the Target (i.e., the real device) is already actively advertising the DFU Service, the App automatically connects to the Target and uploads the latest vendor firmware, fully recovering the device.

3.5 Securing BLE Devices

This section outlines key actionable steps for both vendors and end users to take to reduce the risk of using BLE. These steps would have prevented the firmware attack presented in this work.

3.5.1 Recommendations for Vendors

Use BLE Encryption. Encryption is enabled by the key exchange performed during pairing and bonding. Vendors should force pairing using the *Numerical Comparison* method with Elliptical Curve Diffie-Hellman (ECDH) key exchange. Compatibility issues (e.g. devices support don't support Bluetooth v4.2) can be resolved by using one of the other available pairing methods. In that case, *Out of Band* (OOB) is preferred, as it is more resilient to brute force, albeit more difficult to deploy. Even the vulnerable *Just Works* and *6-digit PIN* methods are preferable over no encryption at all. With encryption enabled, the eavesdropping attacker would not have learned the device password and would not have been able to reverse engineer the application protocol.

Provide Stronger Authentication. Pairing provides a mechanism for device authentication, since LTKs are unique to each pair of devices. Even if data are not encrypted, bonded devices can use the CSRK to sign the data. Vendors should also provide stronger authentication at the application layer, especially if not using BLE encryption. Higher layers could also employ ECDH to exchange encryption or signature keys. Using strong authentication, the Target would have been able to detect that the attacker was not an authorized user and could have ignored the attack commands.

Implement Security in the DFU Service. The default Nordic DFU service does not require authentication to accept new firmware. Vendors should add a layer of protection by requiring a signature with the firmware to determine its authenticity. The device bootloader would need to store the vendor public key from a public-private key pair, which it could use to validate the signature. Vendors should also require physical interaction (e.g., a button press/hold, or a switch flip) with the device to perform a firmware update. While this may make it harder to update firmware on devices located in remote environments, it provides an added layer of security that is difficult to defeat by an attacker without physical access.

3.5.2 Recommendations for End Users

Enable Security Wherever Possible. Users should enable security mechanisms whenever the option is available. This includes, but is not limited to, using passwords, enabling optional pairing, and adding physical security. As an example, the password on the MX1101 caused no hindrance to the user and added a hurdle against an attacker.

Use Risk Management Strategies. Use of BLE devices should be approached with risk management strategies. Users should perform routine security evaluations,

especially on those devices that are part of critical systems. With that information, users can determine the amount of risk they incur by employing these sensors and balance it with how much risk they are willing to accept. For example, while a compromised temperature sensor in a break room poses relatively low risk, a compromised pressure sensor on a manufacturing line can have a much larger impact.

Monitor Devices for Abnormal Behavior. Some devices provide activity logs that can show abnormal behavior to an attentive administrator. Those should be collected and audited with some regularity. Unfortunately, the attack presented in this paper is an example in which logs cannot be trusted. For critical assets, users should deploy BLE sniffers to monitor communications and generate trusted third-party auditable logs. Unfortunately, current sniffers have limitations that make them impractical for some types of applications. Those limitations, along with enhancements to overcome them are presented in Chapter IV.

3.6 Conclusion

This chapter presented the challenges of attack detection on BLE devices. Using an example attack methodology, it illustrated how an industrial monitor is vulnerable to a firmware attack. The monitor cannot detect the attack because it does not require strong authentication, and an end user is unlikely to detect the attack because the malware can behave similarly to the original firmware. The best way to detect an attack is by using a BLE sniffer. While remaining undetectable to the attacker, the sniffer would capture and log the series of ATT commands executed in the attack, and an attentive administrator (or security appliance) could process those logs to detect that the attack took place.

Added layers of security, like encryption at the link layer, stronger authentication and DFU service authentication would have thwarted the attack, but it is the re-

sponsibility of vendors to implement those mechanisms in future applications. In the meantime, end users should use BLE sniffers to audit communications, and approach use of BLE industrial monitors with risk management strategies.

IV. Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections

4.1 Introduction

The previous chapter described how BLE sniffers can be used to detect attacks. Unfortunately, current sniffers have two main limitations that make them impractical for WSNs and other CI applications. First, current sniffers cannot listen to more than one connection at a time. WSNs can employ many simultaneous connections to share information and would therefore require as many sniffers as there are links to cover all communications. A second limitation is that some sniffers fail to maintain synchronization with active connections, which significantly degrades the capture capabilities of the sniffer over time.

The goal of this chapter is to extend BLE sniffer detection capabilities to monitor WSNs by introducing and evaluating BLE-Multi, a sniffer that can reliably capture multiple connections simultaneously. In doing so, the work tackles IQ3 and IQ4, which focus on enhancing BLE sniffers to better protect WSNs and performing evaluations of available sniffers. More specifically, this chapter achieves the following contributions: (i) improved connection synchronization over other open-source sniffer platforms, (ii) novel multi-connection tracking scheme and implementation, (iii) baseline evaluation of frame capture rates when sniffing multiple connections and (iv) comparison of commercially-available and open source sniffers.

The contents of this chapter have been submitted for publication in the proceedings of the Eleventh Annual IFIP WG 11.10 Conference on Critical Infrastructure Protection [24]. This chapter is structured as follows: Section 4.2 provides an overview of commercially available sniffers. Section 4.3 addresses general design goals and considerations for BLE-Multi. Next, Section 4.4 outlines the process to follow and

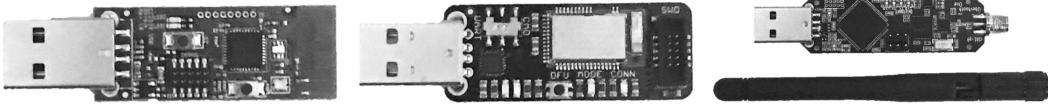
synchronize to a single connection, and Section 4.5 presents the process for tracking more than one connection. Section 4.6 evaluates BLE-Multi against commercially-available sniffers and forms a baseline for frame capture rates when sniffing more than one connection, while Section 4.7 provides a top-level analysis of evaluation results as they relate to auditing.

4.2 Problems with Current BLE Sniffers

Figure 12 shows commonly used BLE sniffers. The TI CC2540 and the Bluefruit LE are pre-loaded with proprietary sniffer firmware and are designed to aid with application debugging. Note that Nordic Semiconductor provides sniffer firmware that can be installed on any BLE radio chip in the nRF51 family [36], to include the nRF51 Development Kit (DK) [37] and the nRF51822 chip [38] used in the Adafruit Bluefruit LE platform. This work focuses only on the Adafruit Bluefruit LE as a representation of Nordic Semiconductor sniffer firmware.

As an open source BLE sniffer, the Ubertooth One can be easily flashed with the latest firmware from the Ubertooth online code repository [18] or with custom firmware for specialized applications. BLE-Multi extends the latest Ubertooth One firmware to achieve synchronized sniffing of multiple connections.

Table 4 highlights the key differences and limitations of the three currently available sniffers, as compared to BLE-Multi. While capabilities vary across sniffers, the single limitation shared among all currently available sniffers is the restriction to monitor a single active BLE connection. When using a sniffer as traffic analysis tool for debugging or troubleshooting, this restriction makes sense because it maximizes the probability of capturing every frame in the conversation. However, when using sniffers under a risk management framework or for a security appliance, it may be acceptable to sacrifice the probability of frame capture in exchange for other more



(a) TI CC2540

(b) Bluefruit LE

(c) Ubertooth One

Figure 12. Commercially-available Bluetooth Low Energy sniffers.

Table 4. Comparison of commercially-available sniffers and BLE-Multi.

Sniffer	Platform	Firmware	Connections
TI CC2540 [52]	Windows	Proprietary	One, synchronized
Bluefruit LE [3]	Windows/Linux	Proprietary	One, synchronized
Ubertooth One [42]	Linux	Open source	One, unsynchronized
BLE-Multi	Linux	Open source	Multiple, synchronized

important capabilities.

A second limitation affecting only the Ubertooth One is that its current firmware does not employ mechanisms to maintain synchronization with an active connection. Connection synchronization is necessary because clocks on different devices can drift from each other. Because BLE communications require interactions on specific frequencies at specific times, the BLE specification provides a procedure to account for clock drift. Unfortunately, the procedure relies on frame directionality, which is not inherent through frame inspection, and is therefore difficult to extract by a sniffer.

This work removes these limitations by proposing firmware for a multi-connection BLE traffic capture platform and a novel adaptation of the BLE synchronization procedure to enhance the Ubertooth One.

4.3 Design Considerations

This section discusses the assumptions and considerations that drive the design of BLE-Multi as an extension to the Ubertooth One firmware.

4.3.1 General Considerations

The goal of a sniffer is to capture every frame sent between the master and slave. Recall that the master controls the link layer parameters that control the connection. Thus, in order to follow a connection, a sniffer can simply act like the slave, capturing and interpreting link layer control messages from the master. As a relayer of data, a sniffer is not required to process data intended for higher layers in the BLE stack; instead, it can indiscriminately forward those data to the host or end user.

One unique situation arises with the encryption procedure. When a master or slave enables encryption, both sides use a symmetric 128-bit key and AES-128 block cypher to perform encryption at the link layer. Traffic captured by a sniffer, including link control messages, would be incomprehensible to the sniffer and end user. This work focuses on securing devices that do not employ encryption and therefore assumes that neither master or slave enables encryption.

Another key assumption is that a sniffer is a passive third-party participant in the conversation. This assumption implies that the sniffer cannot transmit packets to alter the conversation in any way. For example, it cannot request packet re-transmissions, negotiate more convenient connection parameters or otherwise actively interfere with the connection.

An additional implication of its third-party status is that a sniffer cannot easily determine the directionality of an observed frame because frames do not encode sender or receiver information. Such information is obvious in a two-party conversation and is therefore omitted (i.e. if the slave receives a frame, it must have come from the master and vice versa). A passive third-party listener cannot make the same inference. This creates problems with the BLE synchronization mechanism because it relies on the timing of anchor frames sent by the master. Section 4.4.1 addresses this issue by proposing a synchronization mechanism that relies on consecutive empty frames.

Finally, this work does not seek to overcome physical limitations imposed by radio frequency (RF) communications. It does not address radio factors that affect how well it can decode bit sequences from RF energy, nor does it address the effects of interference and noise on the crowded 2.4 GHz ISM band. It abstracts factors like signal-to-noise ratio and bit errors as an overall degradation in the likelihood of frame capture.

4.3.2 Ubertooth One Platform Considerations

This work builds BLE-Multi as an extension to the current Ubertooth One firmware for two reasons. First, Ubertooth One is a flexible open-source hardware platform that lends itself for rapid proof-of-concept development. Second, the Ubertooth One is widely used in the security community because it is one of the only affordable means to capture Bluetooth Classic traffic. Thus, development on this platform can generate meaningful impact to the security community.

The current firmware, named *bluetooth-rxtx*, is capable of sniffing Bluetooth Classic and BLE connections [18]. Its main mode of operation relies on clock interrupts to drive channel hops and radio retuning. Although the available clock resolution for timestamps is 100 ns, clock interrupts run 3,125 times slower, at a frequency of 3,200 Hz. While it is possible to increase its frequency by writing to registers on the on-board CC2400 radio [51], this work seeks to have minimal impact on all other functionality of *bluetooth-rxtx*. Hence, BLE-Multi is restricted to a clock interrupt precision of 312,500 ns. To account for the relatively low precision, BLE-Multi makes conservative approximations when calculating its deadlines and timers. For example, when calculating the timer related to a connection interval, BLE-Multi rounds down to the nearest factor of 312,500 ns, ensuring that the radio performs a hop before the connection interval actually begins. In contrast, when calculating the timer related

to the length of a connection event, BLE-Multi rounds up, conservatively extending the amount of time the radio spends on the channel.

One final implementation consideration of the Ubertooth One is that frame timestamps are taken after a frame is completely received, but many calculations in BLE require knowledge about the timing of the first bit in the frame. Adding to the problem, it takes time for the radio front end to capture the frame and forward it to the processor. Since there is currently no built-in method to know when a packet is first received, BLE-Multi subtracts a constant from every frame timestamp. This constant encapsulates the frame transmission delay and any other processing delays, and it was experimentally tuned throughout the firmware development process.

4.4 Following a Single Connection

In theory, a sniffer need only act like a slave to follow an active connection. In practice, a sniffer does not have the benefit of active participation in a connection. Although this limitation removes its responsibilities in link layer control, it consequently limits its capability to synchronize to a connection. This section highlights how BLE-Multi addresses connection synchronization.

4.4.1 Synchronization

Clock synchronization is essential to BLE communications because BLE employs frequency hopping triggered on time intervals. Poor synchronization can have adverse effects on sniffing, as described by the following example.

Suppose the master has clock drift of 250 parts per million (ppm), normal for a standard Broadcom/Cypress BRCM20702 BLE transceiver [11] used in many commercial BLE dongles. The 16 MHz oscillator in the Ubertooth One is advertised to have a clock drift of 20 ppm [2]. In the worst case, after just 45 seconds, the clocks

on the Ubertooth One and the master could have drifted a full 12.15 ms from each other. A normal frame is transmitted in less than 150 μ s, meaning that the sniffer would have potentially missed some (or all) frames across multiple connection events.

The following subsections describe the built-in BLE synchronization mechanism and the adaptations made in BLE-Multi to combat this problem.

4.4.1.1 Synchronization in BLE

The BLE specification addresses the issue of clock drift with a synchronization mechanism based on the timing of messages from the master. Using the connection request, the master informs the slave of its Sleep Clock Accuracy (SCA), or clock drift. Once a slave receives an anchor (i.e., first packet sent by master), it predicts when the next anchor will arrive using the connection interval:

$$\text{nextExpectedAnchor} = \text{lastAnchor} + \text{connectionInterval}$$

Because the master and slave clocks can drift apart, the slave calculates a window of time during which it should listen for the next anchor. Since the slave knows its own SCA (SCA_S) and the SCA of the master (SCA_M), it can calculate the window of time for the next anchor based on the maximum amount of time the clocks could have drifted. The BLE specification defines this as window widening [7]:

$$\text{windowWidening} = \frac{SCA_M + SCA_S}{1,000,000} * (\text{nextExpectedAnchor} - \text{lastAnchor})$$

As illustrated in Figure 13, the slave listens for *windowWidening* before and *windowWidening* after *nextExpectedAnchor*. Each time a new anchor arrives, the slave resets *lastAnchor*, narrowing *windowWidening*. Note that it is the responsibility of the slave, not the master, to maintain synchronization. By extension, it is the

responsibility of the sniffer to maintain synchronization with the master. However, this presents unique challenges for third-party listeners.

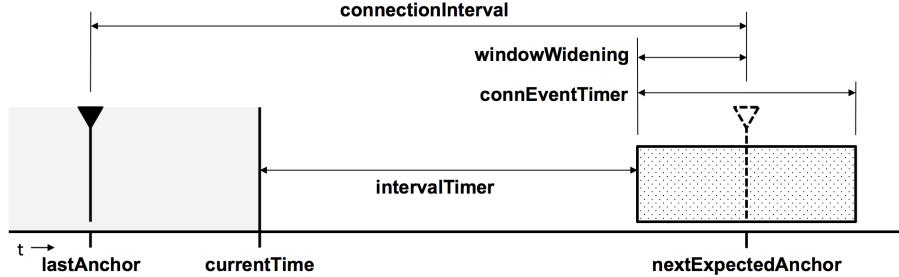


Figure 13. Relationship between anchors, connection parameters and BLE-Multi timers.

4.4.1.2 Synchronization in Sniffers

Since the slave synchronizes to the master, the sniffer does not need to know SCA_S of the actual slave; it simply needs to use its own sleep clock accuracy ($SCA_{Ubertooth}$). However, as previously discussed, anchors are not marked in any way. The sniffer does not know and cannot infer whether a frame came from the master or the slave based strictly on frame content. Clearly, this becomes a problem when calculating `windowWidening`.

One approach is to assume the first frame received by the sniffer is an anchor. However, if that frame is not actually an anchor, the sniffer incorrectly approximates `nextExpectedAnchor`, causing problems on successive CEs. Another approach is to follow frame sequence numbers and make guesses about missed or dropped packets, but such an approach is not practical for this work. BLE-Multi could deliberately choose not to listen to a CE in lieu of another connection, causing the sniffer to lose track of sequence numbers.

While it may not be possible to determine an anchor for every CE, one specific sequence of frames guarantees the presence of an anchor. If two frames are received consecutively (per [7], within 150 μs of each other), and they both contain no data

and have the More Data (MD) bit set to 0, then the first frame is an anchor. Claim 1 presents a proof of this assertion.

Claim 1. If two consecutive frames, f_1 and f_2 , have no data and $MD = 0$, then f_1 is an anchor.

Proof. Suppose that f_1 is not an anchor. By definition, this means that at least one frame was transmitted before f_1 . Let the frame that was transmitted immediately before f_1 be f_0 . MD_{f_0} is either 0 or 1:

- *Case 1:* $MD_{f_0} = 0$. In this case, the sender of f_0 will not send any more data frames. Since f_1 has no data and $MD_{f_1} = 0$, both parties will interpret f_1 as the end of the connection event, and neither side will send any more frames. However, this contradicts the assumption that f_2 was sent. It follows that *Case 1* cannot occur based on the BLE specification [7].
- *Case 2:* $MD_{f_0} = 1$. In this case, the sender of f_0 will transmit more data the next time it sends a frame. Since the master and slave alternate sending frames, the data will be transmitted in the slot for f_2 . But, this contradicts the assumption that f_2 has no data. It follows that *Case 2* cannot occur based on [7].

Since neither *Case 1* nor *Case 2* can occur, f_0 could not have been transmitted. Therefore, f_1 must be the first frame in the communication, and by definition, an anchor. \square

Under this assumption, BLE-Multi cannot extract an anchor from every connection event like a slave would be able to do. However, the sequence of frames described above occurs frequently while the link layer awaits data from higher layers. It is therefore a feasible mechanism to maintain connection synchronization. When the sniffer observes the sequence of empty frames, it updates *lastAnchor*, which automatically

narrows *windowWidening*.

In practice, BLE-Multi applies synchronization by using two timers that decrement with every clock interrupt: (i) *intervalTimer* and (ii) *connEventTimer*. When *intervalTimer* expires, BLE-Multi triggers the BLE radio to hop onto and begin listening on the next channel in the hopping scheme. At that point, the sniffer recalculates *windowWidening* and updates *intervalTimer* as:

$$\textit{intervalTimer} = \textit{nextExpectedAnchor} - \textit{windowWidening} - \textit{currentTime}$$

and it enables *connEventTimer*, which is set to:

$$\textit{connEventTimer} = 2 * \textit{windowWidening}$$

The end of *connEventTimer* signifies the latest time at which the next anchor could arrive. If at any point BLE-Multi receives a frame before *connEventTimer* expires, it assigns 150 μs to *connEventTimer*, which is the maximum amount of time without transmission before a connection event must end.

4.4.2 Link Layer Control

Acting as a slave, a sniffer must appropriately react to link layer control frames, specifically those that dictate or affect connection parameters. Tables 5 and 6 list critical commands sent on advertisement and data channels, respectively, along with actions required by the sniffer. Note that a sniffer only needs to take action on 3 of 7 types of frames sent on advertisement channels and 3 of 22 types sent on data channels. These six types of frames are critical because they signal changes to the underlying connection. To clarify, this does not mean that the sniffer does not capture other types of frames – it does. It simply does not take any action.

Table 5. Critical advertisement channel messages and their effects on BLE connections.

Message Type	Effect on Connection
ADV_IND	Advertiser is connectable Action: Capture CONNECT_REQ.
ADV_DIRECT_IND	Advertiser is connectable Action: Capture CONNECT_REQ.
CONNECT_REQ	Sender requests connection initiation with preceding advertiser Action: Extract connection parameters and actively follow connection

Table 6. Critical data channel control messages and their effects on BLE connections.

Message Type	Effect on Connection
LL_CONNECTION_UPDATE_REQ	Master dictates a change in connection parameters Action: Apply new connection parameters
LL_CHANNEL_MAP_REQ	Master dictates a change in channel map Action: Apply new channel map
LL_TERMINATE_IND	Sender terminates the connection Action: Stop following the connection

4.5 Following Multiple Connections

A sniffer can take advantage of the characteristics of BLE connections to opportunistically follow multiple conversations. This section shows how BLE-Multi is structured to accomplish this task.

4.5.1 Procedure

Figure 3 showed the empty space after a CE that can be used to listen to other connections. By simultaneously tracking the state of multiple active connections, BLE-Multi can choose a different connection after a CE ends. Figure 14 illustrates how the empty space between CEs can be utilized more efficiently by a sniffer cap-

turing three simultaneous connections. CEs on the left side of the Scheduler have already been captured, and CEs on the right side are expected to occur in the future.

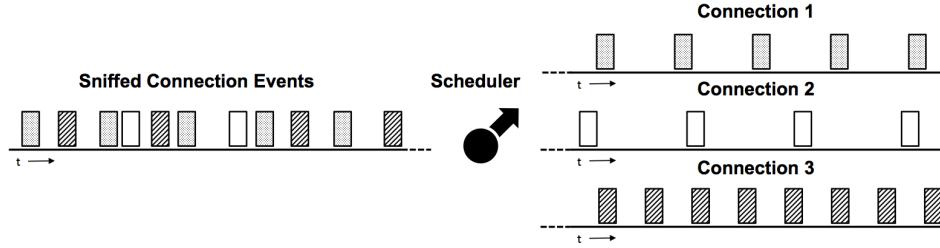


Figure 14. Connection Event Scheduler in BLE-Multi.

Algorithm 1: SCHEDULE. Determine the next link to capture based on the time remaining before the next Connection Event.

```

1: procedure SCHEDULE(activeLinks)
   Input: the current set of active links being tracked, activeLinks
   Output: the next link to capture
2:   minTimer  $\leftarrow \infty$ 
3:   minLink  $\leftarrow \text{NULL}$ 
4:   for link  $\in$  activeLinks do            $\triangleright$  Find the link with minimum timer.
5:     if link.intervalTimer  $<$  minTimer then
6:       minLink  $\leftarrow$  link
7:       minTime  $\leftarrow$  link.intervalTimer
8:     if minTimer  $\geq$  MIN_ADV_TIME then  $\triangleright$  If there's enough time to listen
                                         for advertisements, do so.
9:       return advertisementLink
10:    else
11:      return minLink
12: end

```

At the end of each CE, the Scheduler selects the next CE to capture using the procedure in Algorithm 1. BLE-Multi also employs an *advertisementLink*, which it treats as a default link. If there are no active links, or if there is still enough time before the next CE, the Scheduler selects the *advertisementLink*, which is always configured to track an advertisement channel. This is how the sniffer adds new connections to the set of active links.

Currently, the Scheduler selects the link with the most imminent CE (i.e., the link with minimum *intervalTimer*). However, the sniffer is not limited in the selection algorithm it uses. For example, another implementation may select the link that has been active the longest (oldest first) or the link that has been captured least recently (longest since last capture). Each resulting algorithm would have unique implications on the data capture capabilities for each link.

4.5.2 Limitations

To be clear, the physical limitation of the BLE radio still exists: the sniffer is only capable of listening to one channel at a time. However, because CEs do not normally utilize the entire Connection Interval, BLE-Multi is able to timeshare across multiple connections. This is in contrast to [41], which performs wideband sniffing of Bluetooth Classic communications by capturing multi-channel chunks of the 2.4-GHz spectrum and extracting the Bluetooth conversation via post-processing. This physical limitation of sniffing one channel at a time manifests itself when sniffing multiple connections with multiple masters.

If active connections are initiated by separate masters, there is a probability that CEs in separate connections will overlap. BLE-Multi is forced to choose one connection over the other, and the overall frame capture rate may drop. By contrast, connections that share a master are optimal for BLE-Multi because the master automatically deconflicts CEs. In that case, events do not overlap, and BLE-Multi can listen to a greater number of events. However, even if BLE-Multi encounters multiple connections with multiple masters, CEs often consist of empty frames (no data); this was the insight that allowed BLE-Multi to synchronize with active connections. Hence, missed CEs do not necessarily imply missed data.

Finally, the master may change connection parameters in the middle of the con-

versation. When listening to multiple active connections, BLE-Multi has greater probability of missing a parameter update, which could cause the sniffer to fall out of synchronization with the corresponding link.

4.6 Evaluation

Sniffer evaluation is inherently difficult because there is no mechanism to extract the number of link layer frames transmitted or received by transceivers in BLE dongles. Ideally, evaluation would consist of comparing the number of frames captured by the sniffer with the number of frames actually transmitted. Unfortunately, dongles hide low-level communications from the host, including empty frames and link control frames, disqualifying the most obvious metric of percentage of frames captured. Instead, the experiments in this work target the reception of usable and actionable data embodied by a single command sent from the master to the slave at a random time point after connection start.

4.6.1 Experiment Design

The three experiments in this work use the same experiment setup, procedure and data analysis techniques. The following paragraphs highlight the generic design of these experiments.

Setup. Figure 15 provides an overview of the experiment setup. A laptop with a 2.4-GHz Intel Core i7 processor, 8 GB RAM, and Kali Rolling 2016.1 OS, controls the timing and data collection for each experiment. The sniffers under test (SUT) are connected to a powered USB 2.0 Hub that is connected directly to the laptop. Two BLE dongles (D1 and D2) with BCM20702 transceivers [11] are connected to an additional powered USB 2.0 Hub, which is also connected to the control laptop. D1 and D2 are placed 30 cm away from one Onset MX1101 Temperature/Humid-

ity sensor and one August Smartlock. These two targets represent different classes of critical infrastructure applications: environmental sensing and building security. Additionally, these targets do not implement short timeouts on active connections, meaning that each trial can run for over 45 seconds without receiving an automatic disconnect from the target.

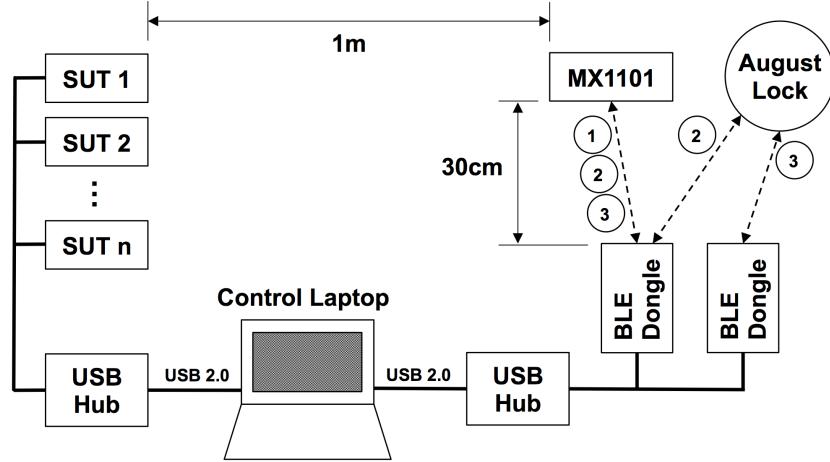


Figure 15. Generic experiment setup (numbered bubbles indicate active links for Experiments 1, 2 and 3).

Figure 15 also illustrates the active links for each of the three experiments. Experiment 1 only establishes the link between a single dongle and a single target ($D1 \rightarrow MX1101$), Experiment 2 connects a single dongle to two targets ($D1 \rightarrow MX1101$ and $D1 \rightarrow August$), and Experiment 3 uses two separate dongles to connect to two targets (one each) ($D1 \rightarrow MX1101$ and $D2 \rightarrow August$).

Trials. This work aggregates trials to extract the likelihood of a frame being captured at increasing times in the BLE connection. Data points at increasing times enable validation of the synchronization techniques introduced by this work. A Python script controls the experiments, where each trial consists of the following:

1. Establish required connections (shown in Figure 15) in pseudorandom order using `random.randint()` in Python.

2. Wait for a pseudorandomly-selected delay of 0, 15, 30, or 45 seconds.
3. Send a single target frame with byte sequence 0x010108010855 to each slave in the order connections were established.
4. Disconnect from each slave in the order connections were established.

Table 7 summarizes the control, independent, and dependent variables in each experiment. Experiment 1 runs 500x4 trials, while Experiments 2 and 3 run 150x4 trials due to time limitations. A trial is deemed successful if the sniffer captures the target frame with the specified byte sequence. Because trials are not conducted in a Faraday cage, electromagnetic interference could impact the capture success of the SUTs. To reduce the effects of unpredictable interference for any particular time delay, the message delay for each trial is determined pseudo-randomly.

Table 7. Summary of control, independent and dependent variables for experiments.

Control Variables	
Distance, BLE dongles and targets	30 cm
Distance, BLE sniffers and active links	1 m
Sniffers under test	TI CC2540, Bluefruit LE, Uberooth One, BLE-Multi
Number of trials	4x500 (Exp 1), 4x150 (Exp 2/3) (due to time)
Independent Variables	
Number of active links	1 (Exp 1), 2 (Exp 2/3)
Configuration of active links	D1→MX1101 (Exp 1), D1→MX1101 and D1→August (Exp 2), D1→MX1101 and D2→August (Exp 3)
Time of ATT message in each trial	0, 15, 30, or 45 sec
Dependent Variables	
Capture success/failure of ATT message by each sniffer, per trial	

Calculating Success Proportion. Trials are aggregated to calculate a success proportion for each time delay. The number of trials in an experiment, n , represents

a single population sample of length n . The sample proportion, \hat{p}_i is calculated using the number of successful captures s_i at time delay $i \in \{0, 15, 30, 45\}$ and n :

$$\hat{p}_i = \frac{s_i}{n_i}$$

The resulting \hat{p}_i is used as an unbiased estimator for the proportion to calculate a 95% confidence interval:

$$\hat{p}_i \pm 1.96 \left(\sqrt{\frac{\hat{p}_i(1 - \hat{p}_i)}{n}} \right)$$

The calculated confidence interval represents how well a sniffer can capture frames at time delay i . Performing this calculation across four points in the connection (0, 15, 30 and 45 seconds) shows how the effectiveness of each sniffer changes with time.

4.6.2 Experiment 1: One Master to One Slave

This experiment compares success proportions for four different SUTs over the length of a single active connection. The SUTs are the BLE-Multi, the Ubertooth One, the TI 2540 sniffer and the Adafruit Bluefruit LE sniffer. Figure 16 and Table 8 show the results of Experiment 1.

Of note, both the TI 2540 and Bluefruit LE perform the best in capture success and synchronization. Both are closed-source commercial sniffers that use hardware specifically designed for BLE communications. This is in contrast to the open-source Ubertooth One (and by extension, BLE-Multi), which tunes an older generic 2.4-GHz radio not specifically designed for BLE. The difference in hardware could account for the gap in performance between the open-source and commercial sniffers. However, this work uses the Ubertooth One platform because of its flexibility for rapid proof-of-concept development and its impact to the security community.

The decreasing trend line on the success proportion of the Ubertooth One is

Table 8. Raw results for Experiment 1 (one master to one slave).

TI CC2540				Bluefruit LE			
i	Successes	\hat{p}_i	95% CI	Successes	\hat{p}_i	95% CI	
0	499/500	0.998	± 0.004	494/500	0.988	± 0.010	
15	498/500	0.996	± 0.006	498/500	0.996	± 0.006	
30	500/500	1.000	-	498/500	0.996	± 0.006	
45	497/500	0.994	± 0.007	496/500	0.992	± 0.008	

Ubertooth One				BLE-Multi			
i	Successes	\hat{p}_i	95% CI	Successes	\hat{p}_i	95% CI	
0	469/500	0.938	± 0.021	420/500	0.840	± 0.032	
15	409/500	0.818	± 0.034	412/500	0.824	± 0.033	
30	362/500	0.724	± 0.039	411/500	0.822	± 0.034	
45	314/500	0.628	± 0.042	430/500	0.860	± 0.030	

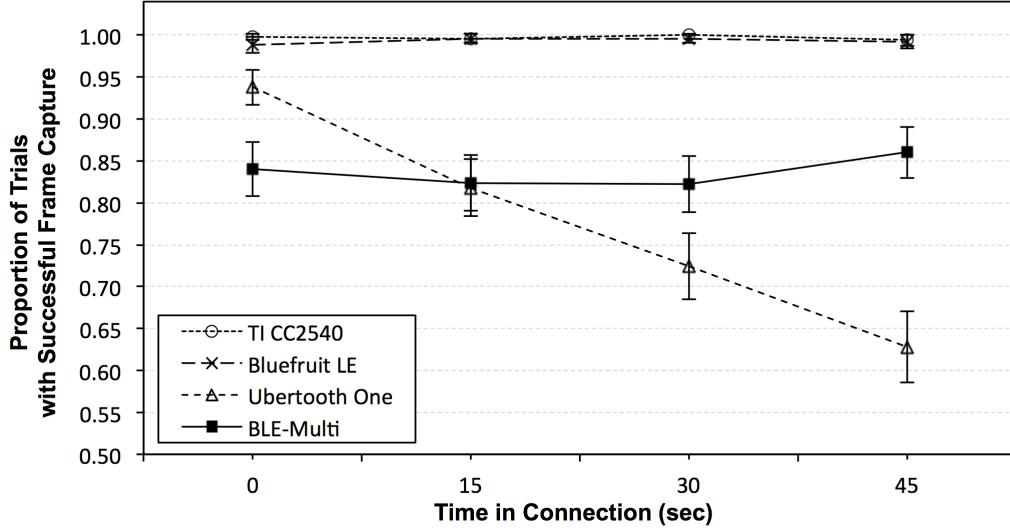


Figure 16. Frame capture success of a single connection over time (95% CI for proportion).

likely due to the lack of connection synchronization mechanism. Conversely, the non-decreasing trend line on the BLE-Multi data suggests that the adapted synchronization mechanism that this work introduces achieves synchronization over the length of the connection. An apparent side effect of BLE-Multi is that while it achieves overall synchronization and stability, its capture success is 5-15% lower than the Ubertooth

One for connections lasting less than 15 seconds. This could be due to the active use of *advertisementLink*, where the sniffer moves away from an active connection to listen to an advertisement channel, or it could stem from an expanded processor payload as a result of the added logic. There may be ways to optimize BLE-Multi to surpass the Ubertooth One, but this work focuses instead on developing the proof-of-concept mechanisms to perform synchronization and capture multiple connections.

4.6.3 Experiment 2: One Master to Multiple Slaves

This experiment tests the effectiveness of BLE-Multi on two simultaneous connections between a single master and multiple slaves. Using the generic setup, the

Table 9. Raw results for Experiment 2 (one master to two slaves).

MX1101 Sensor				August Smartlock			
i	Successes	\hat{p}_i	95% CI	Successes	\hat{p}_i	95% CI	
0	129/150	0.860	± 0.056	122/150	0.813	± 0.062	
15	133/150	0.887	± 0.051	128/150	0.853	± 0.057	
30	130/150	0.867	± 0.054	122/150	0.813	± 0.062	
45	127/150	0.847	± 0.058	120/150	0.800	± 0.064	

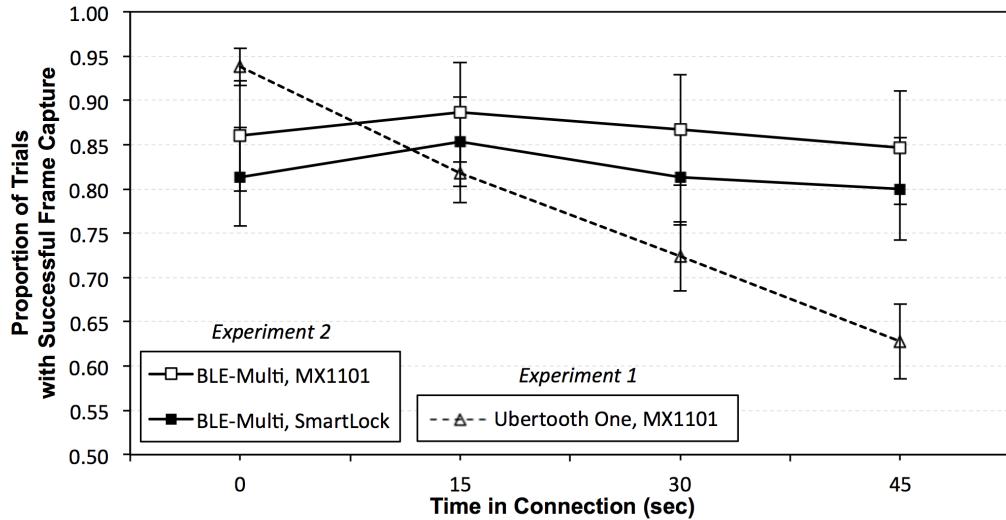


Figure 17. Frame capture success of two simultaneous connections with one master (95% CI for proportion).

experiment uses a single SUT with BLE-Multi, a single BLE dongle and both targets.

A single master connected to two slaves automatically deconflicts both connections by ensuring no overlap between connection events. In other words, these two connections are optimal for simultaneous capture because BLE-Multi never has to miss a connection event in favor of another. Table 9 shows the raw results from the experiment, while Figure 17 illustrates the same results and compares them to the proportion of successful trials of the Ubertooth One in Experiment 1.

The stable BLE-Multi data trends for the MX1101 and August SmartLock show that BLE-Multi is successfully able to capture and maintain synchronization with both connections simultaneously. Notably, on long connections, BLE-Multi on multiple connections performs better than Ubertooth One on a single connection.

4.6.4 Experiment 3: Multiple Masters to Multiple Slaves

This experiment tests the effectiveness of BLE-Multi on two simultaneous connections between two masters and two slaves (one slave per master). Employing the generic setup, the experiment uses a single SUT with BLE-Multi, two BLE dongles and both targets. Figure 18 and Table 10 show the results of Experiment 3. The results for the Ubertooth One from Experiment 1 are again added for reference.

Use of two master-slave pairs adds the possibility of overlapping connection events. In that case, BLE-Multi could be forced to deliberately ignore a connection event and potentially miss data. However, the results here show that, as in Experiment 2, BLE-Multi is able to capture data from and maintain synchronization with both connections simultaneously. Under the conditions in this experiment, overlapping events did not have a statistically significant effect on BLE-Multi capabilities.

One contributing factor is that the connections observed in this experiment are relatively quiet; that is, most CEs consist of only two empty packets. This allows

Table 10. Raw results for Experiment 3 (two masters to two slaves, one each).

MX1101 Sensor				August Smartlock		
i	Successes	\hat{p}_i	95% CI	Successes	\hat{p}_i	95% CI
0	130/150	0.867	± 0.054	126/150	0.840	± 0.059
15	123/150	0.820	± 0.061	128/150	0.853	± 0.057
30	135/150	0.900	± 0.048	127/150	0.847	± 0.058
45	128/150	0.853	± 0.057	134/150	0.893	± 0.049

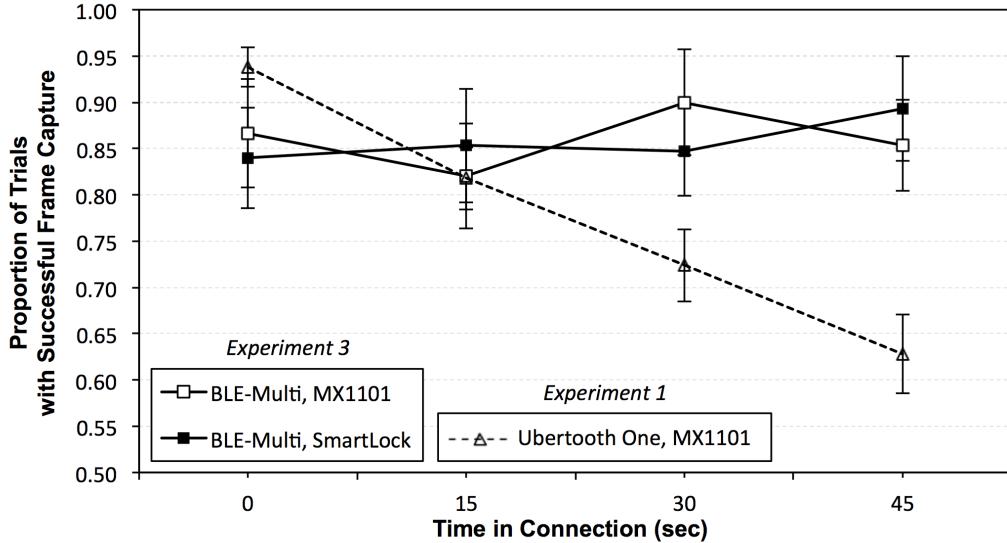


Figure 18. Frame capture success of two simultaneous connections with two masters (95% CI for proportion).

the sniffer more time between CEs and consequently, more time to listen to other connections. Following more than two connections, or more active connections, would likely yield a lower proportion of successful trials per connection.

4.7 Analysis

Results from Experiments 2 and 3 support the claim that BLE-Multi is capable of following and capturing more than one simultaneous BLE connection. Furthermore, the non-decreasing trend lines in all BLE-Multi data support the claim that the synchronization mechanism introduced in this work, which relies on frequent transmissions of empty frames, is a practical solution for long-lived connections. Note that

both proprietary sniffers are also able to synchronize to long-lived connections, but BLE-Multi can do so for multiple connections at the same time.

Because of its synchronization mechanism, BLE-Multi (on multiple connections) can capture frames better than the Ubertooth One (on a single connection) for connections lasting longer than 15 seconds. However, on connections lasting less than 15 seconds, BLE-Multi is 5-15% less likely than the Ubertooth One to capture frames. Since BLE-Multi is an extension to Ubertooth One firmware, the drop in trial success is likely due to the added workload required to capture multiple connections. Even then, BLE-Multi captured target frames with a capture success of more than 70% in all evaluated cases.

Admittedly, while these proportions of successful frame captures may be acceptable for security applications that aim to generally bolster security, they may not be acceptable for applications that require greater scrutiny of traffic. Ideally, each of the simultaneous BLE connections would be captured by one commercial sniffer, maximizing the likelihood of frame capture. Unfortunately, such a system of sniffers may be difficult to scope, design and deploy, as the number of active connections in an environment would not be known in advance. Cost-benefit analyses should be performed for specific applications to determine whether BLE-Multi is an acceptable traffic capture solution. In the end, BLE-Multi provides security professionals the flexibility necessary to audit evolving BLE environments by capturing more than one active conversation at a time while simultaneously listening for new connections.

4.8 Conclusion

This chapter presented BLE-Multi, a firmware update to the Ubertooth One platform that can capture multiple BLE connections simultaneously. Code for BLE-Multi firmware is included in Appendix A and should be used in conjunction with

the other files in the Ubertooth One repository. BLE-Multi was evaluated against other commercially available sniffers on a single active connection, where it successfully demonstrated synchronization to long-lived connections and outperformed other open-source sniffers. BLE-Multi was also evaluated on multiple connections with varying master-slave configurations. Even under the most challenging configuration (multiple masters to multiple slaves), BLE-Multi successfully captured a majority of the traffic in both connections, while outperforming an Ubertooth One tuned to a single connection.

BLE-Multi is a viable auditing solution for CI applications, including WSNs, that use many communication links simultaneously. A single BLE-Multi sniffer could track and listen to multiple active connections and create logs that an administrator or security appliance can audit for malicious or unusual activity. While those logs by themselves would not provide active defense of deployed sensors, they are a step closer to automated WSN/BLE defensive tools.

V. Future Work and Conclusion

This chapter highlights the key contributions of this work and lays out areas for future research, placing emphasis on further enhancing and employing BLE-Multi to provide more robust defensive measures. The work focused on the research question, “**Can BLE traffic sniffers be used to enhance CI security?**,” by individually addressing four investigative questions. Contributions presented in this thesis support the hypothesis that the security of CI and WSNs can be enhanced with a BLE sniffer that detects attacks on multiple simultaneous connections.

IQ1: What vulnerabilities exist on BLE devices used in CI? Chapter III demonstrated an attack on a temperature/humidity HVAC monitor that exposed its vulnerabilities and exploited them to download malware onto the monitor. Unfortunately, the level of security in BLE devices depends on the implementation by the manufacturer. Vendors should enforce pairing/binding to enable device authentication and link encryption, but they often choose to leave the link layer unprotected. Users should approach BLE usage with risk management strategies, minimizing risk by enabling as much optional security as possible, and balancing the amount of risk they take with the amount of risk they are willing to accept.

IQ2: Can BLE sniffers detect attacks against BLE devices? Chapter III also presented the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer. The peripheral is not well-positioned to detect an attack because it cannot differentiate between an attacker and a legitimate user, and the end user is unlikely to detect an attack if it does not visibly affect the device functionality. By contrast, the BLE sniffer provides enough visibility into traffic (i.e., shows ATT commands) to expose anomalous behavior to an administrator or security appliance.

IQ3: Can current BLE sniffers be altered to capture multiple connections to enhance their impact to WSNs? Current traffic sniffers can only follow one connection at a time, making them impractical for applications that employ more than one simultaneous connection. Additionally, some open-source sniffers lack the functionality to synchronize to active connections, causing a drastic drop in capture success over the length of a connection. Chapter IV introduced BLE-Multi, which successfully extended sniffer capabilities to capture and synchronize with more than one simultaneous connection. This upgrade directly enhances the impact of BLE sniffers on WSN security.

IQ4: How can the performance of BLE sniffers be measured? The most natural way to evaluate sniffers is to compare the total number of sniffed frames with the total number transmitted between devices. Unfortunately, BLE transceivers do not provide insight into the total number of frames they exchange with other devices, making it inherently difficult to evaluate sniffers. This work presented an evaluation technique that targeted the reception of usable data, embodied by single ATT Write Command sent from master to slave. The evaluation showed that BLE-Multi could successfully capture multiple connections, maintaining synchronization throughout the connection and outperforming other open-source sniffers.

5.1 Use Case for BLE-Multi

A facility manager deploys an Ubertooth One sniffer with BLE-Multi firmware in a small building to monitor its HVAC wireless sensors. BLE-Multi captures communications to and from all of the sensors in the building, even if the sensors communicate simultaneously. At the end of the day, the administrator downloads and audits the security logs from BLE-Multi, paying particular attention to ATT Read Requests and ATT Write Requests. She finds that sometime in the afternoon, a device connected

to one of the temperature/humidity monitors and interacted directly with the over-the-air firmware update service implemented by the monitor. The logs show that the device downloaded a large file onto the monitor, after which the monitor started to advertise unusual data. She immediately removes the sensor from operation and alerts security personnel of the attack.

In the future, an IPS could automatically perform this process. The IPS would continuously monitor traffic to and from any devices of interest, searching for any known attack patterns. For example, if the IPS detects interactions with the firmware update service, it could begin jamming the connection, disconnecting the would-be attacker. At the same time, the IPS would send email alerts to security personnel, highlighting the attack traffic that triggered the response.

5.2 Future Work

Future research can focus on tuning BLE-Multi to improve frame capture, as well as automating auditing and detection mechanisms to provide more robust defensive measures. The following subsections capture some of the future work to be done in BLE/WSN security.

5.2.1 Vulnerability Assessments

While this work provides an example of the types of vulnerabilities found in BLE devices, it is not an absolute enumeration of all types of vulnerabilities. Research should continue to thoroughly assess devices to better understand the risks of using BLE as well as the potential threat vectors that attackers can use. This information will continue to inform the development of defensive security tools.

5.2.2 Sniffer Tuning

BLE-Multi uses several constant parameters that affect timing calculations. These parameters can be experimentally tuned to improve sniffer reliability. Additionally, the period between clock interrupts can be shortened to increase the precision of *intervalTimer* and *connEventTimer*. A more precise clock will enable tighter windows around connection events and will allow more time for capturing other links. Once tuned, future work can form more practical comparisons between BLE-Multi and commercial sniffers, specifically in the areas of advertisement and connection request capture (i.e. how good is BLE-Multi at hopping onto a new connection?).

5.2.3 Capability Expansion

The BLE advertisement mechanism allows a slave to use between one and three advertisement channels. The slave sends an advertisement in each advertisement channel, in order of increasing frequency, and a master is allowed up to $150\ \mu s$ to respond. A tuned, more precise, BLE-Multi should be able to follow the advertising device as it moves through the advertisement channels, increasing the probability of capturing a connection request, and consequently, the probability of following a connection.

Another area of capability expansion is in the selection algorithms employed by the Scheduler. Other algorithms may prioritize links that have been active the longest (oldest first), while others may select links that have been “waiting” the longest to capture (longest since last capture). Different algorithms would have different implications on how well BLE-Multi captures data.

5.2.4 Mobile Masters and Slaves

Mobile masters and slaves pose a challenge to traffic capture tools because the relative position and orientation between the target transceivers and sniffer antenna are constantly changing. With the wide availability of BLE in mobile devices (e.g., smartphones and tablets), future research should analyze the impact to BLE-Multi from mobile attackers and mobile targets.

5.2.5 Intrusion Detection System (IDS)

WSN IDSs can be challenging to deploy because they normally require a significant power draw from the sensors themselves [49]. BLE-Multi provides the traffic visibility necessary to perform intrusion detection without drawing additional power from the sensor network. Specifically, the following attacks may be detected by sniffers:

- **Man in the Middle.** If the IDS detects that BLE-Multi follows two distinct active connections to the same “target” address, an attacker may be placing herself between the real target device and the victim.
- **Password Attack.** The IDS may correlate multiple connection events corresponding to the same connection to determine that an attacker is attempting a dictionary or brute force password attack.
- **Firmware Attacks.** An IDS filter on data higher in the BLE protocol stack, including ATT Read and Write commands. Commands to the firmware update services described in Chapter III would indicate a firmware change.
- **Rogue Devices.** The BLE protocol requires that all connectable targets advertise their presence. Thus, an IDS could register when an unknown device begins advertising in its proximity.

5.2.6 Intrusion Protection System (IPS)

Using a jam-based denial mechanism, similar to that shown in [14], an IDS could actively block connections it determines to be malicious, turning an IDS to an IPS.

5.3 Conclusion

BLE sniffers can be used to enhance the security of CI applications by exposing the traffic generated by an attack. For WSNs and other applications that require capture of multiple simultaneous links, BLE-Multi is a viable monitoring solution. A single BLE-Multi sniffer can track and listen to active conversations and create logs that an administrator can audit for malicious or unusual activity. In the future, those logs could be forwarded to automated defensive solutions that require granular visibility into BLE traffic. Those tools could mirror other defensive systems, like IDSs and IPSs, used in other types of communications networks. In the meantime, end users should approach use of BLE in CI with risk management strategies, understanding that its use comes with a risk that they must be willing to accept.

Appendix A. BLE-Multi Source Code

The following sections include source code for BLE-Multi, a firmware enhancement to the Ubertooth One capture tool for Bluetooth Low Energy traffic. Enhancements from these firmware files include: (i) multi-connection capture capabilities; (ii) enhanced clock synchronization; and (iii) updated host tools to provide command-line interfaces with BLE-Multi. The following files from the Ubertooth One code repository (github.com/greatscottgadgets/ubertooth) are altered and included below:

- firmware/bluetooth_rxtx/bluetooth_le.c
- firmware/bluetooth_rxtx/bluetooth_le.h
- firmware/bluetooth_rxtx/bluetooth-rxtx.c
- firmware/bluetooth_rxtx/ubertooth_clock.h
- firmware/common/ubertooth.h
- host/libubertooth/src/ubertooth_control.c
- host/libubertooth/src/ubertooth_interface.h
- host/libubertooth/src/ubertooth.c
- host/libubertooth/src/ubertooth.h
- host/ubertooth-tools/src/ubertooth-btle.c

1.1 firmware/bluetooth_rxtx/bluetooth_le.c

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2012 Dominic Spill
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #include "bluetooth_le.h"
22
23 extern u8 le_channel_idx;
24 extern u8 le_hop_amount;
25
26 u16 btle_next_hop(le_state_t *le) {
27     u16 phys = btle_channel_index_to_phys(le->channel_idx);
28     le->channel_idx = (le->channel_idx + le->channel_increment) % 37;
29     return phys;
30 }
31
32 u8 btle_channel_index(u8 channel) {
33     u8 idx;
34     channel /= 2;
35     if (channel == 0)
36         idx = 37;
37     else if (channel < 12)
38         idx = channel - 1;
39     else if (channel == 12)
40         idx = 38;
41     else if (channel < 39)
42         idx = channel - 2;
43     else
44         idx = 39;
45     return idx;
46 }
47
48 u8 btle_afh_channel_index(le_state_t *l) {
49     u8 idx = 0;
50
51     /* Is the channel an advertisement channel? */
52     if (l->channel_idx == 37 || l->channel_idx == 38 || \
53         l->channel_idx == 39){ idx = l->channel_idx;}
54
55     /* Is the unmapped channel in use? */
```

```

56     else if (l->channel_map[l->channel_idx] != 0x00) {
57         idx = l->channel_idx;
58     }
59     /* Unmapped channel is not used, let's map it to a used channel */
60     else {
61         int used_channel_idx = l->channel_idx % l->num_used_channels;
62         int search_idx = -1, i = -1;
63         while (search_idx != used_channel_idx && i < 36) {
64             i++;
65             if (l->channel_map[i] != 0x00) search_idx++;
66         }
67         idx = i;
68     }
69     return idx;
70 }
71
72 u16 btle_channel_index_to_phys(u8 idx) {
73     u16 phys;
74     if (idx < 11)
75         phys = 2404 + 2 * idx;
76     else if (idx < 37)
77         phys = 2428 + 2 * (idx - 11);
78     else if (idx == 37)
79         phys = 2402;
80     else if (idx == 38)
81         phys = 2426;
82     else
83         phys = 2480;
84     return phys;
85 }
86
87 /* calculate CRC */
88 /* note 1: crc_init's bits should be in reverse order */
89 /* note 2: output bytes are in reverse order compared to wire */
90 */
91 /*     example output: */
92 /*             0x6ff46e */
93 */
94 /*     bytes in packet will be: */
95 /*     { 0x6e, 0xf4, 0x6f } */
96 */
97 u32 btle_calc_crc(u32 crc_init, u8 *data, int len) {
98     u32 state = crc_init & 0xffffffff;
99     u32 lfsr_mask = 0x5a6000; /* 010110100110000000000000 */
100    int i, j;
101
102    for (i = 0; i < len; ++i) {
103        u8 cur = data[i];
104        for (j = 0; j < 8; ++j) {
105            int next_bit = (state ^ cur) & 1;
106            cur >>= 1;
107            state >>= 1;
108            if (next_bit) {
109                state |= 1 << 23;
110                state ^= lfsr_mask;
111            }
112        }
113    }

```

```

113     }
114
115     return state;
116 }
117
118 /* runs the CRC in reverse to generate a CRCInit */
119 /*
120 /* crc should be big endian */
121 /* the return will be big endian */
122 */
123 u32 btle_reverse_crc(u32 crc, u8 *data, int len) {
124     u32 state = crc;
125     u32 lfsr_mask = 0xb4c000; /* 101101001100000000000000 */
126     u32 ret;
127     int i, j;
128
129     for (i = len - 1; i >= 0; --i) {
130         u8 cur = data[i];
131         for (j = 0; j < 8; ++j) {
132             int top_bit = state >> 23;
133             state = (state << 1) & 0xffffffff;
134             state |= top_bit ^ ((cur >> (7 - j)) & 1);
135             if (top_bit)
136                 state ^= lfsr_mask;
137         }
138     }
139
140     ret = 0;
141     for (i = 0; i < 24; ++i)
142         ret |= ((state >> i) & 1) << (23 - i);
143
144     return ret;
145 }
146
147 u32 btle_crc_lut[256] = {
148     0x000000, 0x01b4c0, 0x036980, 0x02dd40, 0x06d300, 0x0767c0, 0x05ba80,
149     0x040e40, 0x0da600, 0x0c12c0, 0x0ecf80, 0x0f7b40, 0x0b7500, 0x0ac1c0,
150     0x081c80, 0x09a840, 0x1b4c00, 0x1af8c0, 0x182580, 0x199140, 0x1d9f00,
151     0x1c2bc0, 0x1ef680, 0x1f4240, 0x16ea00, 0x175ec0, 0x158380, 0x143740,
152     0x103900, 0x118dc0, 0x135080, 0x12e440, 0x369800, 0x372cc0, 0x35f180,
153     0x344540, 0x304b00, 0x31ffc0, 0x332280, 0x329640, 0x3b3e00, 0x3a8ac0,
154     0x385780, 0x39e340, 0x3ded00, 0x3c59c0, 0x3e8480, 0x3f3040, 0x2dd400,
155     0x2c60c0, 0x2ebd80, 0x2f0940, 0x2b0700, 0x2ab3c0, 0x286e80, 0x29da40,
156     0x207200, 0x21c6c0, 0x231b80, 0x22af40, 0x26a100, 0x2715c0, 0x25c880,
157     0x247c40, 0x6d3000, 0x6c84c0, 0x6e5980, 0x6fed40, 0x6be300, 0x6a57c0,
158     0x688a80, 0x693e40, 0x609600, 0x6122c0, 0x63ff80, 0x624b40, 0x664500,
159     0x67f1c0, 0x652c80, 0x649840, 0x767c00, 0x77c8c0, 0x751580, 0x74a140,
160     0x70af00, 0x711bc0, 0x73c680, 0x727240, 0x7bda00, 0x7a6ec0, 0x78b380,
161     0x790740, 0x7d0900, 0x7cbdc0, 0x7e6080, 0x7fd440, 0x5ba800, 0x5a1cc0,
162     0x58c180, 0x597540, 0x5d7b00, 0x5ccfc0, 0x5e1280, 0x5fa640, 0x560e00,
163     0x57bac0, 0x556780, 0x54d340, 0x50dd00, 0x5169c0, 0x53b480, 0x520040,
164     0x40e400, 0x4150c0, 0x438d80, 0x423940, 0x463700, 0x4783c0, 0x455e80,
165     0x44ea40, 0x4d4200, 0x4cf6c0, 0x4e2b80, 0x4f9f40, 0x4b9100, 0x4a25c0,
166     0x48f880, 0x494c40, 0xda6000, 0xdbd4c0, 0xd90980, 0xd8bd40, 0xdcdb300,
167     0xdd07c0, 0xdfda80, 0xde6e40, 0xd7c600, 0xd672c0, 0xd4af80, 0xd51b40,
168     0xd11500, 0xd0a1c0, 0xd27c80, 0xd3c840, 0xc12c00, 0xc098c0, 0xc24580,
169     0xc3f140, 0xc7ff00, 0xc64bc0, 0xc49680, 0xc52240, 0xcc8a00, 0xcd3ec0,
170     0xcf380, 0xce5740, 0xca5900, 0xcbedc0, 0xc93080, 0xc88440, 0xecf800,

```

```

171 0xed4cc0, 0xef9180, 0xee2540, 0xea2b00, 0xeb9fc0, 0xe94280, 0xe8f640,
172 0xe15e00, 0xe0eac0, 0xe23780, 0xe38340, 0xe78d00, 0xe639c0, 0xe4e480,
173 0xe55040, 0xf7b400, 0xf600c0, 0xf4dd80, 0xf56940, 0xf16700, 0xf0d3c0,
174 0xf20e80, 0xf3ba40, 0xfa1200, 0xfbba6c0, 0xf97b80, 0xf8cf40, 0xfc100,
175 0xfd75c0, 0xffa880, 0xfe1c40, 0xb75000, 0xb6e4c0, 0xb43980, 0xb58d40,
176 0xb18300, 0xb037c0, 0xb2ea80, 0xb35e40, 0xbaf600, 0xbb42c0, 0xb99f80,
177 0xb82b40, 0xbc2500, 0xbd91c0, 0xbf4c80, 0xbef840, 0xac1c00, 0xada8c0,
178 0xaf7580, 0xaec140, 0xaacf00, 0xab7bc0, 0xa9a680, 0xa81240, 0xa1ba00,
179 0xa00ec0, 0xa2d380, 0xa36740, 0xa76900, 0xa6ddc0, 0xa40080, 0xa5b440,
180 0x81c800, 0x807cc0, 0x82a180, 0x831540, 0x871b00, 0x86afc0, 0x847280,
181 0x85c640, 0x8c6e00, 0x8ddac0, 0x8f0780, 0x8eb340, 0x8abd00, 0x8b09c0,
182 0x89d480, 0x886040, 0x9a8400, 0x9b30c0, 0x99ed80, 0x985940, 0x9c5700,
183 0x9de3c0, 0x9f3e80, 0x9e8a40, 0x972200, 0x9696c0, 0x944b80, 0x95ff40,
184 0x91f100, 0x9045c0, 0x929880, 0x932c40
185 };
186
187 /*
188 * Calculate a BTLE CRC one byte at a time. Thanks to Dominic Spill &
189 * Michael Ossmann for writing and optimizing this.
190 *
191 * Arguments: CRCInit, pointer to start of packet, length of packet in
192 * bytes */
193 u32 btle_crcgen_lut(u32 crc_init, u8 *data, int len) {
194     u32 state;
195     int i;
196     u8 key;
197
198     state = crc_init & 0xffffffff;
199     for (i = 0; i < len; ++i) {
200         key = data[i] ^ (state & 0xff);
201         state = (state >> 8) ^ btle_crc_lut[key];
202     }
203     return state;
204 }
```

1.2 firmware/bluetooth_rxtx/bluetooth_le.h

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2012 Dominic Spill
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #include "ubertooth.h"
22
23 /* General BLE parameters */
24 #define BTLE_CHANNELS 40
25 #define ADVERTISING_CHANNELS 3
26 #define DATA_CHANNELS 37
27
28 /* Byte locations of data within every LE packet. */
29 #define ADV_ADDRESS_IDX 0
30 #define HEADER_IDX 4
31 #define DATA_LEN_IDX 5
32 #define DATA_START_IDX 6
33
34 /* Byte locations of data within a CONNECTREQ packet */
35 #define CRC_INIT (2+4+6+6+4)
36 #define WIN_SIZE (2+4+6+6+4+3)
37 #define WIN_OFFSET (2+4+6+6+4+3+1)
38 #define CONN_INTERVAL (2+4+6+6+4+3+1+2)
39 #define CHANNEL_MAP (2+4+6+6+4+3+1+2+2+2+2)
40 #define CHANNEL_INC (2+4+6+6+4+3+1+2+2+2+2+5)
41
42 typedef enum {
43     LINK_INACTIVE,
44     LINK_LISTENING,
45     LINK_CONN_PENDING,
46     LINK_CONNECTED,
47 } link_state_t;
48
49 typedef struct _le_state_t {
50     /* Access Address to filter by */
51     u32 access_address;
52     /* Access address in CC2400 syncword format */
53     u16 synch;
54     /* lower 16 bits thereof */
55     u16 sync1;
```

```

56  /* CrcInit: used to calculate CRC */
57  u32 crc_init;
58  /* bits-reversed version of the previous */
59  u32 crc_init_reversed;
60  /* true to reject packets with bad CRC */
61  int crc_verify;
62  /* current link layer state */
63  link_state_t link_state;
64  /* current channel index */
65  u8 channel_idx;
66  /* amount to hop */
67  u8 channel_increment;
68
69  /* unused if 0, else used */
70  u8 channel_map[37];
71  /* total number of used channels */
72  u8 num_used_channels;
73
74  /* reference time for the start of the connection */
75  u32 conn_epoch;
76  /* number of intervals remaining before next hop */
77  u32 volatile interval_timer;
78  /* connection-specific hop interval */
79  u32 conn_interval;
80  /* number of intervals since the start of the connection */
81  int volatile conn_count;
82  /* Master's sleep clock accuracy in PPM
83  * (20, 30, 50, 75, 100, 150, 250, 500) */
84  u16 sca;
85
86  /* window size */
87  u32 win_size;
88  /* offset of first window from start of connection */
89  u32 win_offset;
90
91 /* whether a connection update is pending */
92  int update_pending;
93  /* the connection count when the update takes effect */
94  u16 update_instant;
95  /* the new hop_internal */
96  u32 interval_update;
97  /* the new window size */
98  u32 win_size_update;
99  /* the new window offset */
100 u32 win_offset_update;
101 /* whether a channel map update is pending */
102 int map_update_pending;
103 /* unused if 0, else used */
104 u8 channel_map_update[37];
105 /* total number of used channels in the updated map */
106 u8 num_used_channels_update;
107 /* target MAC for connection following (byte order reversed) */
108 u8 target[6];
109 /* whether a target has been set (default: false) */
110 int target_set;
111 /* when was the last packet received */
112 u32 last_packet;

```

```

113     /* time of the last event start in CLK100NS precision */
114     u32 last_confirmed_event;
115     /* time of next event start in CLK100NS precision */
116     u32 next_expected_event;
117     /* Amount of time we want to wait on a channel until we
118      * give up in CLK100NS */
119     u32 linger_time;
120     /* Amount of addt'l time we have to listen for to account for
121      * clock skew */
122     u32 window_widening;
123 } le_state_t;
124
125
126 static const u8 whitening[] = {
127     1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1,
128     1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0,
129     0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
130     0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0,
131     1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0,
132     1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0,
133 };
134
135 static const u8 whitening_index[] = {
136     70, 62, 120, 111, 77, 46, 15, 101, 66, 39, 31, 26, 80,
137     83, 125, 89, 10, 35, 8, 54, 122, 17, 33, 0, 58, 115, 6,
138     94, 86, 49, 52, 20, 40, 27, 84, 90, 63, 112, 47, 102
139 };
140
141 static const u8 hop_interval_lut[] = {
142     0, 1, 19, 25, 28, 15, 31, 16, 14, 33, 26, 27, 34, 20, 8,
143     5, 7, 24, 35, 2, 13, 30, 32, 29, 17, 3, 10, 11, 4, 23, 21,
144     6, 22, 9, 12, 18, 36,
145 };
146
147 u16 btle_next_hop(le_state_t *le);
148 u8 btle_afh_channel_index(le_state_t * le);
149 u8 btle_channel_index(u8 channel);
150 u16 btle_channel_index_to_phys(u8 idx);
151 u32 btle_calc_crc(u32 crc_init, u8 *data, int len);
152 u32 btle_reverse_crc(u32 crc, u8 *data, int len);
153 u32 btle_crcgen_lut(u32 crc_init, u8 *data, int len);
154
155 static const u32 whitening_word[40][12] = {
156 {0xc3bcb240, 0x5f4a371f, 0x9a9cf685, 0x44c5d6c1, 0xe1de5920, 0xfa51b8f,
157 0xcd4e7b42, 0x2262eb60, 0xf0ef2c90, 0x57d28dc7, 0x66a73da1, 0x113175b0
    ,},
158 {0xbcb24089, 0x4a371fc3, 0x9cf6855f, 0xc5d6c19a, 0xde592044, 0xa51b8fe1,
159 0x4e7b42af, 0x62eb60cd, 0xef2c9022, 0xd28dc7f0, 0xa73da157, 0x3175b066
    ,},
160 {0x3da157d2, 0x75b066a7, 0x96481131, 0x46e3f877, 0x9ed0abe9, 0xbad83353,
161 0xcb240898, 0xa371fc3b, 0xcf6855f4, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d
    ,},
162 {0x42afa51b, 0x60cd4e7b, 0x902262eb, 0xc7f0ef2c, 0xa157d28d, 0xb066a73d,
163 0x48113175, 0xe3f87796, 0xd0abe946, 0xd833539e, 0x240898ba, 0x71fc3bcb
    ,},
164 {0x3f877964, 0x0abe946e, 0x833539ed, 0x40898bad, 0xfc3bcb2, 0x855f4a37,
165 0xc19a9cf6, 0x2044c5d6, 0x8fe1de59, 0x42afa51b, 0x60cd4e7b, 0x902262eb
    ,},

```

```

166 {0x40898bad, 0x1fc3bcb2, 0x855f4a37, 0xc19a9cf6, 0x2044c5d6, 0x8fe1de59,
167 0x42afa51b, 0x60cd4e7b, 0x902262eb, 0xc7f0ef2c, 0xa157d28d, 0xb066a73d
    },
168 {0xc19a9cf6, 0x2044c5d6, 0x8fe1de59, 0x42afa51b, 0x60cd4e7b, 0x902262eb,
169 0xc7f0ef2c, 0xa157d28d, 0xb066a73d, 0x48113175, 0xe3f87796, 0xd0abe946
    },
170 {0xbe946e3f, 0x3539ed0a, 0x898bad83, 0xc3bcb240, 0x5f4a371f, 0x9a9cf685,
171 0x44c5d6c1, 0xe1de5920, 0xaafa51b8f, 0xcd4e7b42, 0x2262eb60, 0xf0ef2c90
    },
172 {0x3bcb2408, 0xf4a371fc, 0xa9cf6855, 0x4c5d6c19, 0x1de59204, 0xfa51b8fe,
173 0xd4e7b42a, 0x262eb60c, 0x0ef2c902, 0x7d28dc7f, 0x6a73da15, 0x13175b06
    },
174 {0x44c5d6c1, 0xe1de5920, 0xaafa51b8f, 0xcd4e7b42, 0x2262eb60, 0xf0ef2c90,
175 0x57d28dc7, 0x66a73da1, 0x113175b0, 0xf8779648, 0xab946e3, 0x33539ed0
    },
176 {0xc5d6c19a, 0xde592044, 0xa51b8fe1, 0x4e7b42af, 0x62eb60cd, 0xef2c9022,
177 0xd28dc7f0, 0xa73da157, 0x3175b066, 0x77964811, 0xe946e3f8, 0x539ed0ab
    },
178 {0xbad83353, 0xcb240898, 0xa371fc3b, 0xcf6855f4, 0x5d6c19a9, 0xe592044c,
179 0x51b8fe1d, 0xe7b42afa, 0x2eb60cd4, 0xf2c90226, 0x28dc7f0e, 0x73da157d
    },
180 {0xc7f0ef2c, 0xa157d28d, 0xb066a73d, 0x48113175, 0xe3f87796, 0xd0abe946,
181 0xd833539e, 0x240898ba, 0x71fc3bcb, 0x6855f4a3, 0x6c19a9cf, 0x92044c5d
    },
182 {0xb8fe1de5, 0xb42afa51, 0xb60cd4e7, 0xc902262e, 0xdc7f0ef2, 0xda157d28,
183 0x5b066a73, 0x64811317, 0x6e3f8779, 0xed0abe94, 0xad833539, 0xb240898b
    },
184 {0x39ed0abe, 0x8bad8335, 0xbcb24089, 0x4a371fc3, 0x9cf6855f, 0xc5d6c19a,
185 0xde592044, 0xa51b8fe1, 0x4e7b42af, 0x62eb60cd, 0xef2c9022, 0xd28dc7f0
    },
186 {0x46e3f877, 0x9ed0abe9, 0xbad83353, 0xcb240898, 0xa371fc3b, 0xcf6855f4,
187 0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 0xe7b42afa, 0x2eb60cd4, 0xf2c90226
    },
188 {0x33539ed0, 0x0898bad8, 0xfc3bcb24, 0x55f4a371, 0x19a9cf68, 0x044c5d6c,
189 0xfe1de592, 0x2afa51b8, 0x0cd4e7b4, 0x02262eb6, 0x7f0ef2c9, 0x157d28dc
    },
190 {0x4c5d6c19, 0x1de59204, 0xfa51b8fe, 0xd4e7b42a, 0x262eb60c, 0x0ef2c902,
191 0x7d28dc7f, 0x6a73da15, 0x13175b06, 0x87796481, 0xbe946e3f, 0x3539ed0a
    },
192 {0xcd4e7b42, 0x2262eb60, 0xf0ef2c90, 0x57d28dc7, 0x66a73da1, 0x113175b0,
193 0xf8779648, 0xab946e3, 0x33539ed0, 0x0898bad8, 0xfc3bcb24, 0x55f4a371
    },
194 {0xb240898b, 0x371fc3bc, 0xf6855f4a, 0xd6c19a9c, 0x592044c5, 0x1b8fe1de,
195 0x7b42afa5, 0xeb60cd4e, 0x2c902262, 0x8dc7f0ef, 0x3da157d2, 0x75b066a7
    },
196 {0xcf6855f4, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 0xe7b42afa, 0x2eb60cd4,
197 0xf2c90226, 0x28dc7f0e, 0x73da157d, 0x175b066a, 0x79648113, 0x946e3f87
    },
198 {0xb066a73d, 0x48113175, 0xe3f87796, 0xd0abe946, 0xd833539e, 0x240898ba,
199 0x71fc3bcb, 0x6855f4a3, 0x6c19a9cf, 0x92044c5d, 0xb8fe1de5, 0xb42afa51
    },
200 {0x3175b066, 0x77964811, 0xe946e3f8, 0x539ed0ab, 0x98bad833, 0x3bcb2408,
201 0xf4a371fc, 0xa9cf6855, 0x4c5d6c19, 0x1de59204, 0xfa51b8fe, 0xd4e7b42a
    },
202 {0x4e7b42af, 0x62eb60cd, 0xef2c9022, 0xd28dc7f0, 0xa73da157, 0x3175b066,
203 0x77964811, 0xe946e3f8, 0x539ed0ab, 0x98bad833, 0x3bcb2408, 0xf4a371fc
    }

```

```

        },
204 {0xcb240898, 0xa371fc3b, 0xcf6855f4, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d,
205 0xe7b42afa, 0x2eb60cd4, 0xf2c90226, 0x28dc7f0e, 0x73da157d, 0x175b066a
        },
206 {0xb42afa51, 0xb60cd4e7, 0xc902262e, 0xdc7f0ef2, 0xda157d28, 0x5b066a73,
207 0x64811317, 0x6e3f8779, 0xed0abe94, 0xad833539, 0xb240898b, 0x371fc3bc
        },
208 {0x3539ed0a, 0x898bad83, 0xc3bcb240, 0x5f4a371f, 0x9a9cf685, 0x44c5d6c1,
209 0xe1de5920, 0xafa51b8f, 0xcd4e7b42, 0x2262eb60, 0xf0ef2c90, 0x57d28dc7
        },
210 {0x4a371fc3, 0x9cf6855f, 0xc5d6c19a, 0xde592044, 0xa51b8fe1, 0x4e7b42af,
211 0x62eb60cd, 0xef2c9022, 0xd28dc7f0, 0xa73da157, 0x3175b066, 0x77964811
        },
212 {0x371fc3bc, 0xf6855f4a, 0xd6c19a9c, 0x592044c5, 0x1b8fe1de, 0x7b42afa5,
213 0xeb60cd4e, 0x2c902262, 0x8dc7f0ef, 0x3da157d2, 0x75b066a7, 0x96481131
        },
214 {0x48113175, 0xe3f87796, 0xd0abe946, 0xd833539e, 0x240898ba, 0x71fc3bcb,
215 0x6855f4a3, 0x6c19a9cf, 0x92044c5d, 0xb8fe1de5, 0xb42afa51, 0xb60cd4e7
        },
216 {0xc902262e, 0xdc7f0ef2, 0xda157d28, 0x5b066a73, 0x64811317, 0x6e3f8779,
217 0xed0abe94, 0xad833539, 0xb240898b, 0x371fc3bc, 0xf6855f4a, 0xd6c19a9c
        },
218 {0xb60cd4e7, 0x902262e, 0xdc7f0ef2, 0xda157d28, 0x5b066a73, 0x64811317,
219 0x6e3f8779, 0xed0abe94, 0xad833539, 0xb240898b, 0x371fc3bc, 0xf6855f4a
        },
220 {0x2262eb60, 0xf0ef2c90, 0x57d28dc7, 0x66a73da1, 0x113175b0, 0xf8779648,
221 0xabe946e3, 0x33539ed0, 0x0898bad8, 0xfc3bcb24, 0x55f4a371, 0x19a9cf68
        },
222 {0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 0xe7b42afa, 0x2eb60cd4, 0xf2c90226,
223 0x28dc7f0e, 0x73da157d, 0x175b066a, 0x79648113, 0x946e3f87, 0x39ed0abe
        },
224 {0xdc7f0ef2, 0xda157d28, 0x5b066a73, 0x64811317, 0x6e3f8779, 0xed0abe94,
225 0xad833539, 0xb240898b, 0x371fc3bc, 0xf6855f4a, 0xd6c19a9c, 0x592044c5
        },
226 {0xa371fc3b, 0xcf6855f4, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 0xe7b42afa,
227 0x2eb60cd4, 0xf2c90226, 0x28dc7f0e, 0x73da157d, 0x175b066a, 0x79648113
        },
228 {0xde592044, 0xa51b8fe1, 0x4e7b42af, 0x62eb60cd, 0xef2c9022, 0xd28dc7f0,
229 0xa73da157, 0x3175b066, 0x77964811, 0x946e3f8, 0x539ed0ab, 0x98bad833
        },
230 {0xa157d28d, 0xb066a73d, 0x48113175, 0x3f87796, 0xd0abe946, 0xd833539e,
231 0x240898ba, 0x71fc3bcb, 0x6855f4a3, 0x6c19a9cf, 0x92044c5d, 0xb8fe1de5
        },
232 {0x2044c5d6, 0x8fe1de59, 0x42afa51b, 0x60cd4e7b, 0x902262eb, 0xc7f0ef2c,
233 0xa157d28d, 0xb066a73d, 0x48113175, 0x3f87796, 0xd0abe946, 0xd833539e
        },
234 {0x5f4a371f, 0x9a9cf685, 0x44c5d6c1, 0xe1de5920, 0xafa51b8f, 0xcd4e7b42,
235 0x2262eb60, 0xf0ef2c90, 0x57d28dc7, 0x66a73da1, 0x113175b0, 0xf8779648
        },
236 },
237 };
238 /* LE Multi Connection Stuff */
239 typedef enum {
240     ANCHORSEARCH,
241     ANCHORCANDIDATE
242 } anchor_state_t;

```

```
243
244 typedef enum {
245     ADV_SEARCH,
246     ADV_CANDIDATE
247 } adv_state_t;
248
249 #define F_ADVERTISEMENT 0x01
250 #define F_CONN_EVENT      0x02
251 #define F_DECISION        0x04
252 #define F_CONN_UPDATE     0x08
253 #define F_INTERVAL         0x10
```

1.3 firmware/bluetooth_rxtx/bluetooth-rxtx.c

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2010–2013 Michael Ossmann
4   * Copyright 2011–2013 Dominic Spill
5   *
6   * This program is free software; you can redistribute it and/or modify
7   * it under the terms of the GNU General Public License as published by
8   * the Free Software Foundation; either version 2, or (at your option)
9   * any later version.
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; see the file COPYING. If not, write to
18  * the Free Software Foundation, Inc., 51 Franklin Street,
19  * Boston, MA 02110–1301, USA.
20  */
21
22 #include <string.h>
23 #include "ubertooth.h"
24 #include "ubertooth_usb.h"
25 #include "ubertooth_interface.h"
26 #include "ubertooth_rssi.h"
27 #include "ubertooth_cs.h"
28 #include "ubertooth_dma.h"
29 #include "ubertooth_clock.h"
30 #include "bluetooth.h"
31 #include "bluetooth_le.h"
32 #include "cc2400_rangetest.h"
33 #include "ego.h"
34
35 /***** DEFINES *****/
36 #define MIN(x,y) ((x)<(y)?(x):(y))
37 #define MAX(x,y) ((x)>(y)?(x):(y))
38 #define DIVIDE_ROUND(N, D) ((N) + (D)/2) / (D)
39 #define DIVIDE_CEIL(N, D) (1 + ((N-1) / D))
40 #define AA_LIST_SIZE (int) \
    (sizeof(le_promisc.active_aa) / sizeof(le_promisc.active_aa_t))
41 /* Max # of links to follow in multi-link LE mode */
42 #define MAX_LINKS 5
43 #define MAX_INACTIVE_LINK_TIME 5 /* in seconds */
44 #define JAM_COUNT_DEFAULT 40
45 #define MAX_READ_REG 0x2d
46 #define MAX_NUM_TARGETS 1
47 #define MIN_ADV_INTERVALS 3
48 /* # of clkns intervals needed to capture one frame */
49 #define PACKET_OFFSET 2
50 /* # of clkns intervals until we pass 150us */
51 #define TIFS 5 /* tuned to 5 which is >>> 150us */
52
53 /***** GLOBALS *****/
54
```

```

55 volatile u32 debug_clk100ns_count = 0;
56 /* build info */
57 const char compile_info[] = \
58     "ubertooh " GIT_REVISION \
59     " (" COMPILE_BY "@" COMPILE_HOST ") " TIMESTAMP;
60
61 /* hopping stuff */
62 volatile uint8_t hop_mode = HOP_NONE;
63 volatile uint8_t do_hop = 0;
64 volatile uint16_t channel = 2441;
65 volatile uint16_t hop_direct_channel = 0;
66 volatile uint16_t hop_timeout = 158;
67 volatile uint16_t requested_channel = 0;
68 volatile uint16_t saved_request = 0;
69
70 /* bulk USB stuff */
71 volatile uint8_t idle_buf_clkn_high = 0;
72 volatile uint32_t idle_buf_clk100ns = 0;
73 volatile uint16_t idle_buf_channel = 0;
74 volatile uint8_t dma_discard = 0;
75 volatile uint8_t status = 0;
76
77 /* operation mode */
78 volatile uint8_t mode = MODE_IDLE;
79 volatile uint8_t requested_mode = MODE_IDLE;
80 volatile uint8_t jam_mode = JAM_NONE;
81 volatile uint8_t ego_mode = EGO_FOLLOW;
82 volatile uint8_t modulation = MOD_BT_BASIC_RATE;
83
84 /* specan stuff */
85 volatile uint16_t low_freq = 2400;
86 volatile uint16_t high_freq = 2483;
87 volatile int8_t rssi_threshold = -30;
88
89 /* Generic TX stuff */
90 generic_tx_packet tx_pkt;
91
92 /* le stuff */
93 uint8_t slave_mac_address[6] = { 0, };
94 le_state_t le = {
95     .access_address = 0x8e89bed6, /* advertising channel access address
96     */
97     .synch = 0x6b7d,           /* bit-reversed adv channel AA */
98     .syncl = 0x9171,
99     .crc_init = 0x555555,      /* advertising channel CRCInit */
100    .crc_init_reversed = 0xAAAAAA,
101    .crc_verify = 0,
102    .link_state = LINK_INACTIVE,
103    .conn_epoch = 0,
104    .target_set = 0,
105    .last_packet = 0,
106 };
107
108 typedef struct _le_promisc_active_aa_t {
109     u32 aa;
110     int count;
111 } le_promisc_active_aa_t;

```

```

112 typedef struct _le_promisc_state_t {
113     /* LFU cache of recently seen AA's */
114     le_promisc_active_aa_t active_aa[32];
115
116     /* recovering hop interval */
117     u32 smallest_hop_interval;
118     int consec_intervals;
119 } le_promisc_state_t;
120 le_promisc_state_t le_promisc;
121
122 /* LE Multiple Connection Following Stuff */
123 /*
124  * This structure has two timers/timeouts:
125  *     decision_timer - predetermined timeout that forces the handler
126  *                         to "move on"
127  *     conn_event_timer - calculated timeout, occurs when no packet is
128  *                         received for 150 us
129 */
130 typedef struct _le_multi_link_hdlr_t {
131     /* Container to hold all connection states */
132     le_state_t links[MAX_LINKS];
133     /* Generic advertisement listener */
134     le_state_t adv_link;
135     /* Pointer to the connection currently being followed */
136     le_state_t *cur_link;
137     /* Number of targets set by the user */
138     u8 num_targets;
139     /* target MACs for connection following (byte order reversed) */
140     u8 targets[MAX_NUM_TARGETS][6];
141     /* State of the anchor-capturing SM */
142     anchor_state_t cur_anchor_state;
143     /* CLK100NS timestamp of latest candidate packet for anchor */
144     u32 candidate_anchor;
145     /* Enable/disable decision timer */
146     int decision_timer_en;
147     /* # of CLOCK intervals until we have to make a decision */
148     u16 volatile decision_timer;
149     /* Enable/disable the conn_event_timer, useful when listening to ADV
150      */
151     int conn_event_timer_en;
152     /* if 0, the current connection event has invalidated */
153     u16 volatile conn_event_timer;
154     /* Enable/disable advertisement timer */
155     int adv_timer_en;
156     /* If 0, it's time to move to a different adv channel */
157     u16 volatile adv_timer;
158     adv_state_t cur_adv_state;
159     /* target MAC for advertisement following (byte order reversed) */
160     u8 adv_target[6];
161     /* if 1, we should run the scheduler */
162     int scheduler_en;
163 } le_multi_link_hdlr_t;
164 le_multi_link_hdlr_t le_hdlr = {
165     .links = {
166         .access_address = 0x8e89bed6,
167         .synch = 0x6b7d,

```

```

168     .sync1 = 0x9171,
169     .crc_init = 0x555555,
170     .crc_init_reversed = 0xAAAAAA,
171     .crc_verify = 0,
172     .link_state = LINK_INACTIVE,
173     .conn_epoch = 0,
174     .target_set = 0,
175     .last_packet = 0,
176     .channel_map = {0x00,} ,
177     .num_used_channels = 0,
178     .last_confirmed_event = 0,
179     .next_expected_event = 0,
180     .linger_time = 0,
181     .window_widening = 0,
182   } ,},
183   .adv_link = {
184     .access_address = 0x8e89bed6, /* advertising channel AA */
185     .synch = 0x6b7d, /* bit-reversed adv channel AA */
186     .sync1 = 0x9171,
187     .crc_init = 0x555555, /* advertising channel CRCInit */
188     .crc_init_reversed = 0xAAAAAA,
189     .crc_verify = 0,
190     .link_state = LINK_LISTENING,
191     .channel_idx = 37,
192     .channel_increment = 0,
193     .conn_epoch = 0,
194     .target_set = 0,
195     .last_packet = 0,
196   },
197   .cur_link = NULL,
198   .cur_anchor_state = ANCHOR_SEARCH,
199   .candidate_anchor = 0,
200   .decision_timer = 0,
201   .decision_timer_en = 0,
202   .conn_event_timer_en = 0, /* Disabled */
203   .conn_event_timer = 0,
204   .adv_timer_en = 0,
205   .adv_timer = 0,
206   .cur_adv_state = ADV_SEARCH,
207   .scheduler_en = 1,
208 };
209 u8 hop_reason = 0;
210
211 /* LE jamming Stuff*/
212 int le_jam_count = 0;
213
214 /* General BT/LE Stuff */
215 typedef int (*data_cb_t)(char *);
216 data_cb_t data_cb = NULL; /* Pointer to data handler */
217
218 typedef void (*packet_cb_t)(u8 *);
219 packet_cb_t packet_cb = NULL; /* Pointer to frame handler */
220
221 /* Unpacked symbol buffers (two rxbufs) */
222 char unpacked[DMA_SIZE*8*2];
223
224 /***** FUNCTION PROTOTYPES *****/
225 static int enqueue(uint8_t type, uint8_t* buf);

```

```

226 int enqueue_with_ts(uint8_t type, uint8_t* buf, uint32_t ts);
227 int debug_log(u32 message);
228 static int vendor_request_handler(uint8_t request, \
229     uint16_t* request_params, uint8_t* data, int* data_len);
230 void TIMER0_IRQHandler(void);
231 static void msleep(uint32_t millis);
232 void DMA_IRQHandler(void);
233 static void cc2400_idle(void);
234 static void cc2400_rx(void);
235 static void cc2400_rx_sync(u32 sync);
236 static void cc2400_tx_sync(uint32_t sync);
237 void le_transmit(u32 aa, u8 len, u8 *data);
238 void le_jam(void);
239 void hop(void);
240 void bt_stream_rx(void);
241 static uint8_t reverse8(uint8_t data);
242 static uint16_t reverse16(uint16_t data);
243 void br_transmit(void);
244 static void le_set_access_address(u32 aa, le_state_t *conn);
245 void reset_le(le_state_t *conn);
246 void le_multi_scheduler(void);
247 void reset_le_promisc(void);
248 void bt_generic_le(u8 active_mode);
249 void bt_le_sync(u8 active_mode);
250 void bt_multi_le_sync(void);
251 int cb_follow_le(char *unpacked);
252 void connection_follow_cb(u8 *packet);
253 void connection_multi_follow_cb(u8 *packet);
254 void bt_follow_le(void);
255 void bt_multi_follow_le(void);
256 void le_promisc_state(u8 type, void *data, unsigned len);
257 void promisc_recover_hop_increment(u8 *packet);
258 void promisc_recover_hop_interval(u8 *packet);
259 void promisc_follow_cb(u8 *packet);
260 void see_aa(u32 aa);
261 int cb_le_promisc(char *unpacked);
262 void bt_promisc_le(void);
263 void bt_slave_le(void);
264 void rx_generic_sync(void);
265 void rx_generic(void);
266 void tx_generic(void) ;
267 void specan(void);
268 void led_specan(void);
269 void le_multi_reset_interval_timer(le_state_t *link);
270 void le_multi_cleanup(void);
271 void le_multi_update_adv_state(void);
272
273 /***** MAIN *****/
274 int main() {
275     ubertooh_init();
276     clkn_init();
277     ubertooh_usb_init(vendor_request_handler);
278     cc2400_idle();
279
280     while (1) {

```

```

281 handle_usb(clkn);
282 if(requested_mode != mode) {
283     switch (requested_mode) {
284         case MODE_RESET:
285             /* Allow time for the USB command to return correctly */
286             wait(1);
287             reset();
288             break;
289         case MODE_AFH:
290             mode = MODE_AFH;
291             bt_stream_rx();
292             break;
293         case MODE_RX_SYMBOLS:
294             mode = MODE_RX_SYMBOLS;
295             bt_stream_rx();
296             break;
297         case MODE_TX_SYMBOLS:
298             mode = MODE_TX_SYMBOLS;
299             br_transmit();
300             break;
301         case MODE_BT_FOLLOW:
302             mode = MODE_BT_FOLLOW;
303             bt_stream_rx();
304             break;
305         /* Ubertooh Sniffing request */
306         case MODE_BT_FOLLOW_LE:
307             bt_follow_le();
308             break;
309         case MODE_BT_MULTIFOLLOW_LE:
310             bt_multi_follow_le();
311             break;
312         case MODE_BT_PROMISCLE:
313             bt_promisc_le();
314             break;
315         case MODE_BT_SLAVELE:
316             bt_slave_le();
317             break;
318         case MODE_TX_TEST:
319             mode = MODE_TX_TEST;
320             cc2400_txtest(&modulation, &channel);
321             break;
322         case MODE_RANGETEST:
323             mode = MODE_RANGETEST;
324             cc2400_rangetest(&channel);
325             requested_mode = MODE_IDLE;
326             break;
327         case MODE_REPEATERT:
328             mode = MODE_REPEATERT;
329             cc2400_repeater(&channel);
330             break;
331         case MODE_SPECAN:
332             specan();
333             break;
334         case MODE_LED_SPECAN:
335             led_specan();
336             break;
337         case MODE_EGO:

```

```

338         mode = MODE_EGO;
339         ego_main(ego_mode);
340         break;
341     case MODE_RX_GENERIC:
342         mode = MODE_RX_GENERIC;
343         rx_generic();
344         break;
345     case MODE_TX_GENERIC:
346         tx_generic();
347         break;
348     case MODE_IDLE:
349         cc2400_idle();
350         break;
351     default:
352         /* This is really an error state , but what can you do? */
353         break;
354     }
355   }
356 }
357 }

358 **** Function Definitions ****
359 /*
360 * Function: enqueue
361 * _____
362 * Puts something in the DMA buffer to send to host.
363 * type:
364 * buf:
365 */
366 static int enqueue( uint8_t type , uint8_t* buf) {
367     usb_pkt_rx* f = usb_enqueue();
368
369     /* fail if queue is full */
370     if (f == NULL) {
371         status |= FIFO_OVERFLOW;
372         return 0;
373     }
374
375     f->pkt_type = type;
376     if(type == SPECAN) {
377         f->clkn_high = (clkn >> 20) & 0xff;
378         f->clk100ns = CLK100NS;
379     } else {
380         /*f->clkn_high = idle_buf_clkn_high; */
381         /*f->clk100ns = idle_buf_clk100ns; */
382         f->clkn_high = (clkn >> 20) & 0xff;
383         f->clk100ns = CLK100NS;
384         f->channel = (uint8_t)((idle_buf_channel - 2402) & 0xff);
385         f->rssi_min = rssi_min;
386         f->rssi_max = rssi_max;
387         f->rssi_avg = rssi_get_avg(idle_buf_channel);
388         f->rssi_count = rssi_count;
389     }
390
391     memcpy(f->data , buf , DMA_SIZE);
392
393     f->status = status;

```

```

395     status = 0;
396
397     return 1;
398 }
399
400 /*
401 * Function: enqueue_with_ts
402 *
403 * Puts something in the DMA buffer to send to host, include timestamp.
404 * - type:
405 * - buf:
406 * - ts:
407 */
408 int enqueue_with_ts(uint8_t type, uint8_t* buf, uint32_t ts) {
409     usb_pkt_rx* f = usb_enqueue();
410
411     /* fail if queue is full */
412     if (f == NULL) {
413         status |= FIFO_OVERFLOW;
414         return 0;
415     }
416
417     f->clk_n_high = 0;
418     f->clk100ns = ts;
419     f->channel = (uint8_t)((channel - 2402) & 0xff);
420     f->rssi_avg = 0;
421     f->rssi_count = 0;
422     memcpy(f->data, buf, DMA_SIZE);
423
424     f->status = status;
425     status = 0;
426
427     return 1;
428 }
429
430 /*
431 * Function: debug_log
432 *
433 * Send debug messages back to host via DMA queue. Message shows up as
434 * the access address of the packet. The DMA queue must already be
435 * initialized for this to work!
436 * - message:
437 */
438 int debug_log(u32 message) {
439     uint8_t type = LE_PACKET;
440
441     usb_pkt_rx* f = usb_enqueue();
442
443     /* fail if queue is full */
444     if (f == NULL) {
445         status |= FIFO_OVERFLOW;
446         return 0;
447     }
448
449     f->pkt_type = type;
450     f->clk_n_high = idle_buf_clk_n_high;
451     f->clk100ns = idle_buf_clk100ns;
452     f->channel = (uint8_t)((idle_buf_channel - 2402) & 0xff);
453     f->rssi_min = rssi_min;

```

```

454     f->rssi_max = rssi_max;
455     f->rssi_avg = rssi_get_avg(idle_buf_channel);
456     f->rssi_count = rssi_count;
457     uint32_t* p = (uint32_t*) f->data;
458     p[0] = message;
459     return 1;
460 }
461 /*
462 * Function: vendor_request_handler
463 */
464 *
465 * Respond to requests from host to set global parameters and start
466 * sniffing.
467 * - request:
468 * - request_params:
469 * - data:
470 * - data_len:
471 */
472 static int vendor_request_handler(uint8_t request, \
473     uint16_t* request_params, uint8_t* data, int* data_len) {
474
475     uint32_t command[5];
476     uint32_t result[5];
477     uint64_t ac_copy;
478     uint32_t clock;
479     size_t length; /* string length */
480     usb_pkt_rx* p = NULL;
481     uint16_t reg_val;
482     uint8_t i;
483
484     switch (request) {
485     case UBERTOOTH_PING:
486         *data_len = 0;
487         break;
488     case UBERTOOTH_RX_SYMBOLS:
489         requested_mode = MODE_RX_SYMBOLS;
490         *data_len = 0;
491         break;
492     case UBERTOOTH_TX_SYMBOLS:
493         hop_mode = HOP_BLUETOOTH;
494         requested_mode = MODE_TX_SYMBOLS;
495         *data_len = 0;
496         break;
497     case UBERTOOTH_GETUSRLED:
498         data[0] = (USRLED) ? 1 : 0;
499         *data_len = 1;
500         break;
501     case UBERTOOTH_SETUSRLED:
502         if (request_params[0])
503             USRLED_SET;
504         else
505             USRLED_CLR;
506         break;
507     case UBERTOOTH_GETRXLED:
508         data[0] = (RXLED) ? 1 : 0;
509         *data_len = 1;
510         break;
511     case UBERTOOTH_SETRXLED:

```

```

512     if (request_params[0])
513         RXLED_SET;
514     else
515         RXLED_CLR;
516     break;
517 case UBERTOOTH_GET_TXLED:
518     data[0] = (TXLED) ? 1 : 0;
519     *data_len = 1;
520     break;
521 case UBERTOOTH_SET_TXLED:
522     if (request_params[0])
523         TXLED_SET;
524     else
525         TXLED_CLR;
526     break;
527 case UBERTOOTH_GET_1V8:
528     data[0] = (CC1V8) ? 1 : 0;
529     *data_len = 1;
530     break;
531 case UBERTOOTH_SET_1V8:
532     if (request_params[0])
533         CC1V8_SET;
534     else
535         CC1V8_CLR;
536     break;
537 case UBERTOOTH_GET_PARTNUM:
538     get_part_num(data, data_len);
539     break;
540 case UBERTOOTH_RESET:
541     requested_mode = MODE_RESET;
542     break;
543 case UBERTOOTH_GET_SERIAL:
544     get_device_serial(data, data_len);
545     break;
546
547 #ifdef UBERTOOTH_ONE
548 case UBERTOOTH_GET_PAEN:
549     data[0] = (PAEN) ? 1 : 0;
550     *data_len = 1;
551     break;
552 case UBERTOOTH_SET_PAEN:
553     if (request_params[0])
554         PAEN_SET;
555     else
556         PAEN_CLR;
557     break;
558 case UBERTOOTH_GET_HGM:
559     data[0] = (HGM) ? 1 : 0;
560     *data_len = 1;
561     break;
562 case UBERTOOTH_SET_HGM:
563     if (request_params[0])
564         HGM_SET;
565     else
566         HGM_CLR;
567     break;
568#endif
569

```

```

570 #ifdef TX_ENABLE
571     case UBERTOOTH_TX_TEST:
572         requested_mode = MODE_TX_TEST;
573         break;
574     case UBERTOOTH_GET_PALEVEL:
575         data[0] = cc2400_get(FREND) & 0x7;
576         *data_len = 1;
577         break;
578     case UBERTOOTH_SET_PALEVEL:
579         if( request_params[0] < 8 ) {
580             cc2400_set(FREND, 8 | request_params[0]);
581         } else {
582             return 0;
583         }
584         break;
585     case UBERTOOTH_RANGE_TEST:
586         requested_mode = MODE_RANGE_TEST;
587         break;
588     case UBERTOOTH_REPEATERT:
589         requested_mode = MODE_REPEATERT;
590         break;
591 #endif
592
593     case UBERTOOTH_RANGE_CHECK:
594         data[0] = rr.valid;
595         data[1] = rr.request_pa;
596         data[2] = rr.request_num;
597         data[3] = rr.reply_pa;
598         data[4] = rr.reply_num;
599         *data_len = 5;
600         break;
601     case UBERTOOTH_STOP:
602         requested_mode = MODE_IDLE;
603         break;
604     case UBERTOOTH_GET_MOD:
605         data[0] = modulation;
606         *data_len = 1;
607         break;
608     case UBERTOOTH_SET_MOD:
609         modulation = request_params[0];
610         break;
611     case UBERTOOTH_GET_CHANNEL:
612         data[0] = channel & 0xFF;
613         data[1] = (channel >> 8) & 0xFF;
614         *data_len = 2;
615         break;
616     case UBERTOOTH_SET_CHANNEL:
617         requested_channel = request_params[0];
618         /* bluetooth band sweep mode, start at channel 2402 */
619         if (requested_channel > MAX_FREQ) {
620             hop_mode = HOP_SWEEP;
621             requested_channel = 2402;
622         }
623         /* fixed channel mode, can be outside bluetooth band */
624         else {
625             hop_mode = HOP_NONE;
626             requested_channel = MAX(requested_channel, MIN_FREQ);

```

```

627     requested_channel = MIN( requested_channel , MAX_FREQ) ;
628 }
629
630 if (mode != MODE_BT_FOLLOWLE && mode != MODE_BT_MULTIFOLLOWLE) {
631     channel = requested_channel ;
632     requested_channel = 0;
633
634     /* CS threshold is mode-dependent. Update it after
635      * possible mode change. TODO - klugy. */
636     cs_threshold_calc_and_set(channel);
637 }
638 break;
639 case UBERTOOTH_SET_ISP:
640     set_isp();
641     *data_len = 0; /* should never return */
642     break;
643
644     bootloader_ctrl = DFU_MODE;
645     requested_mode = MODE_RESET;
646     break;
647 case UBERTOOTH_SPECAN:
648     if (request_params[0] < 2049 || request_params[0] > 3072 ||
649         request_params[1] < 2049 || request_params[1] > 3072 ||
650         request_params[1] < request_params[0])
651         return 0;
652     low_freq = request_params[0];
653     high_freq = request_params[1];
654     requested_mode = MODE_SPECAN;
655     *data_len = 0;
656     break;
657 case UBERTOOTH_RX_GENERIC:
658     requested_mode = MODE_RX_GENERIC;
659     *data_len = 0;
660     break;
661 case UBERTOOTH_LED_SPECAN:
662     if (request_params[0] > 256)
663         return 0;
664     rssi_threshold = 54 - request_params[0];
665     requested_mode = MODE_LED_SPECAN;
666     *data_len = 0;
667     break;
668 case UBERTOOTH_GET_REV_NUM:
669     data[0] = 0x00;
670     data[1] = 0x00;
671
672     length = (u8)strlen(GIT_REVISION);
673     data[2] = length;
674
675     memcpy(&data[3], GIT_REVISION, length);
676
677     *data_len = 2 + 1 + length;
678     break;
679 case UBERTOOTH_GET_COMPILE_INFO:
680     length = (u8)strlen(compile_info);
681     data[0] = length;
682     memcpy(&data[1], compile_info, length);
683     *data_len = 1 + length;
684     break;

```

```

685     case UBERTOOTH_GET_BOARD_ID:
686         data[0] = BOARD_ID;
687         *data_len = 1;
688         break;
689     case UBERTOOTH_SET_SQUELCH:
690         cs_threshold_req = (int8_t)request_params[0];
691         cs_threshold_calc_and_set(channel);
692         break;
693     case UBERTOOTH_GET_SQUELCH:
694         data[0] = cs_threshold_req;
695         *data_len = 1;
696         break;
697     case UBERTOOTH_SET_BDADDR:
698         target.address = 0;
699         target.syncword = 0;
700         for(int i=0; i < 8; i++) {
701             target.address |= (uint64_t)data[i] << 8*i;
702         }
703         for(int i=0; i < 8; i++) {
704             target.syncword |= (uint64_t)data[i+8] << 8*i;
705         }
706         precalc();
707         break;
708     case UBERTOOTH_START_HOPPING:
709         clk_n_offset = 0;
710         for(int i=0; i < 4; i++) {
711             clk_n_offset <= 8;
712             clk_n_offset |= data[i];
713         }
714         hop_mode = HOP_BLUETOOTH;
715         dma_discard = 1;
716         DIO_SSEL_SET;
717         clk100ns_offset = (data[4] << 8) | (data[5] << 0);
718         requested_mode = MODE_BT_FOLLOW;
719         break;
720     case UBERTOOTH_AFH:
721         hop_mode = HOP_AFH;
722         requested_mode = MODE_AFH;
723
724         for(int i=0; i < 10; i++) {
725             afh_map[i] = 0;
726         }
727         used_channels = 0;
728         afh_enabled = 1;
729         break;
730     case UBERTOOTH_HOP:
731         do_hop = 1;
732         break;
733     case UBERTOOTH_SET_CLOCK:
734         clock = data[0] | data[1] << 8 | data[2] << 16 | data[3] << 24;
735         clk_n = clock;
736         cs_threshold_calc_and_set(channel);
737         break;
738     case UBERTOOTH_SET_AFHMAP:
739         for(int i=0; i < 10; i++) {
740             afh_map[i] = data[i];
741         }

```

```

742     afh_enabled = 1;
743     *data_len = 10;
744     break;
745 case UBERTOOTH_CLEAR_AFHMAP:
746     for( int i=0; i < 10; i++) {
747         afh_map[ i ] = 0;
748     }
749     afh_enabled = 0;
750     *data_len = 10;
751     break;
752 case UBERTOOTH_GET_CLOCK:
753     clock = clk_n;
754     for( int i=0; i < 4; i++) {
755         data[ i ] = (clock >> (8*i)) & 0xff;
756     }
757     *data_len = 4;
758     break;
759 case UBERTOOTH_TRIM_CLOCK:
760     clk100ns_offset = (data[0] << 8) | (data[1] << 0);
761     break;
762 case UBERTOOTH_FIX_CLOCK_DRIFT:
763     clk_drift_ppm += (int16_t)(data[0] << 8) | (data[1] << 0);
764
765     /* Too slow */
766     if (clk_drift_ppm < 0) {
767         clk_drift_correction = 320 / (uint16_t)(-clk_drift_ppm);
768         clk_n_next_drift_fix = clk_n_last_drift_fix +
769         clk_drift_correction;
770     }
771     /* Too fast */
772     else if (clk_drift_ppm > 0) {
773         clk_drift_correction = 320 / clk_drift_ppm;
774         clk_n_next_drift_fix = clk_n_last_drift_fix +
775         clk_drift_correction;
776     }
777     /* Don't trim */
778     else {
779         clk_drift_correction = 0;
780         clk_n_next_drift_fix = 0;
781     }
782     break;
783 case UBERTOOTH_BTLE_SNIFFING:
784     *data_len = 0;
785
786     do_hop = 0;
787     hop_mode = HOP_BTLE;
788     requested_mode = MODE_BT_FOLLOW_LE;
789
790     queue_init();
791     cs_threshold_calc_and_set(channel);
792     break;
793 case UBERTOOTH_BTLE_MULTI_SNIFFING:
794     *data_len = 0;
795
796     do_hop = 0;
797     hop_mode = HOP_BTLE_MULTI;
798     requested_mode = MODE_BT_MULTIFOLLOW_LE;

```

```

797     queue_init();
798     cs_threshold_calc_and_set(channel);
799     break;
800
801 case UBERTOOTH_GET_ACCESS_ADDRESS:
802     for(int i=0; i < 4; i++) {
803         data[i] = (le.access_address >> (8*i)) & 0xff;
804     }
805     *data_len = 4;
806     break;
807 case UBERTOOTH_SET_ACCESS_ADDRESS:
808     le_set_access_address(data[0] | data[1] << 8 | \
809     data[2] << 16 | data[3] << 24, &le);
810     le.target_set = 1;
811     break;
812 case UBERTOOTH_DO_SOMETHING:
813     /* do something! just don't commit anything here */
814     break;
815 case UBERTOOTH_DO_SOMETHING_REPLY:
816     /* after you do something, tell me what you did! */
817     /* don't commit here please */
818     data[0] = 0x13;
819     data[1] = 0x37;
820     *data_len = 2;
821     break;
822 case UBERTOOTH_GET_CRC_VERIFY:
823     data[0] = le.crc_verify ? 1 : 0;
824     *data_len = 1;
825     break;
826 case UBERTOOTH_SET_CRC_VERIFY:
827     le.crc_verify = request_params[0] ? 1 : 0;
828     break;
829 case UBERTOOTH_POLL:
830     p = dequeue();
831     if (p != NULL) {
832         memcpy(data, (void *)p, sizeof(usb_pkt_rx));
833         *data_len = sizeof(usb_pkt_rx);
834     } else {
835         data[0] = 0;
836         *data_len = 1;
837     }
838     break;
839 case UBERTOOTH_BTLE_PROMISC:
840     *data_len = 0;
841
842     hop_mode = HOP_NONE;
843     requested_mode = MODE_BT_PROMISC_LE;
844
845     queue_init();
846     cs_threshold_calc_and_set(channel);
847     break;
848 case UBERTOOTH_READ_REGISTER:
849     reg_val = cc2400_get(request_params[0]);
850     data[0] = (reg_val >> 8) & 0xff;
851     data[1] = reg_val & 0xff;
852     *data_len = 2;
853     break;

```

```

854     case UBERTOOTH_WRITE_REGISTER:
855         cc2400_set(request_params[0] & 0xff, request_params[1]);
856         break;
857     case UBERTOOTH_WRITE_REGISTERS:
858         for(i=0; i<request_params[0]; i++) {
859             reg_val = (data[(i*3)+1] << 8) | data[(i*3)+2];
860             cc2400_set(data[i*3], reg_val);
861         }
862         break;
863     case UBERTOOTH_READ_ALL_REGISTERS:
864         for(i=0; i<=MAX_READ_REG; i++) {
865             reg_val = cc2400_get(i);
866             data[i*3] = i;
867             data[(i*3)+1] = (reg_val >> 8) & 0xff;
868             data[(i*3)+2] = reg_val & 0xff;
869         }
870         *data_len = MAX_READ_REG*3;
871         break;
872     case UBERTOOTH_TX_GENERIC_PACKET:
873         i = 7 + data[6];
874         memcpy(&tx_pkt, data, i);
875         /*tx_pkt.channel = data[4] << 8 | data[5]; */
876         requested_mode = MODE_TX_GENERIC;
877         *data_len = 0;
878         break;
879     case UBERTOOTH_BTLE_SLAVE:
880         memcpy(slave_mac_address, data, 6);
881         requested_mode = MODE_BT_SLAVE;
882         break;
883     case UBERTOOTH_BTLE_SET_TARGET:
884         /* Addresses appear in packets in reverse-octet order. */
885         /* Store the target address in reverse order so that */
886         /* we can do a simple memcmp later */
887         le.target[0] = data[5];
888         le.target[1] = data[4];
889         le.target[2] = data[3];
890         le.target[3] = data[2];
891         le.target[4] = data[1];
892         le.target[5] = data[0];
893         le.target_set = 1;
894         break;
895
896 #ifdef TX_ENABLE
897     case UBERTOOTH_JAM_MODE:
898         jam_mode = request_params[0];
899         break;
900 #endif
901
902     case UBERTOOTHLEGO:
903 #ifndef TX_ENABLE
904         if (ego_mode == EGO_JAM)
905             return 0;
906 #endif
907         requested_mode = MODE_EGO;
908         ego_mode = request_params[0];
909         break;
910     case UBERTOOTH_GET_API_VERSION:

```

```

911     for (int i = 0; i < 4; ++i)
912         data[i] = (UBERTOOTH_APLVERSION >> (8*i)) & 0xff;
913     *data_len = 4;
914     break;
915
916     default:
917         return 0;
918     }
919     return 1;
920 }
921
922 /*
923 * Function: TIMER0_IRQHandler
924 */
925 * Handle interrupts triggered by the on-board clock. This function
926 * serves as the "driver" for following connections by using timeouts.
927 * Updates CLKN.
928 */
929 void TIMER0_IRQHandler() {
930     if (T0IR & TIR_MR0_Interrupt) {
931         debug_clk100ns_count++;
932
933         clkn += clkn_offset + 1;
934         clkn_offset = 0;
935
936         uint32_t le_clk = (clkn - le.conn_epoch) & 0x03;
937
938         /* Trigger hop based on mode */
939         /* NONE or SWEEP -> 25 Hz */
940         if (hop_mode == HOP_NONE || hop_mode == HOP_SWEEP) {
941             if ((clkn & 0x7f) == 0)
942                 do_hop = 1;
943         }
944
945         /* BLUETOOTH -> 1600 Hz */
946         else if (hop_mode == HOP_BLUETOOTH) {
947             if ((clkn & 0x1) == 0)
948                 do_hop = 1;
949         }
950
951         /* BLUETOOTH Low Energy -> 7.5ms - 4.0s in multiples of 1.25 ms */
952         else if (hop_mode == HOP_BTLE) {
953             /* Only hop if connected */
954             if (le.link_state == LINK_CONNECTED && le_clk == 0) {
955                 --le.interval_timer;
956
957                 /* We reached the # of intervals until a hop is necessary */
958                 if (le.interval_timer == 0) {
959                     do_hop = 1;
960                     ++le.conn_count;
961                     le.interval_timer = le.conn_interval; /* Reset intervals
962
963                     */
964                 } else {
965                     TXLED_CLR; /* hack! */
966                 }
967             }
968         }
969     }

```

```

968 /* Multi-connection tracking. Here's where we detect that a
969 * connection event ends */
970 else if (hop_mode == HOP_BTLE_MULTI ) {
971     /* Decrease the advertisement timer, trigger if enabled */
972     if (le_hdlr.adv_timer_en == 1){
973         --le_hdlr.adv_timer;
974         if (le_hdlr.adv_timer == 0) {
975             le_multi_update_adv_state();
976             do_hop = 1;
977             hop_reason |= FADVERTISEMENT;
978         }
979     }
980
981     /* Decrease the connection event timer, trigger if enabled */
982     if (le_hdlr.conn_event_timer_en == 1){
983         --le_hdlr.conn_event_timer;
984         if (le_hdlr.conn_event_timer == 0) {
985             do_hop = 1;
986             hop_reason |= FCONN_EVENT;
987         }
988     }
989
990     /* Decrease the decision timer, trigger if enabled */
991     if (le_hdlr.decision_timer_en == 1){
992         --le_hdlr.decision_timer;
993         if (le_hdlr.decision_timer == 0) {
994             do_hop = 1;
995             hop_reason |= FDECISION;
996         }
997     }
998
999     /* Progress all connection interval timers */
1000    /* If an interval timer expires, we simply increase the
1001     * connection count and reset the interval timer. If the timer
1002     * changed our current link, we'll have to make sure we hop to
1003     * a new channel.
1004 */
1005    le_state_t *l = NULL;
1006    for ( int i = 0; i < MAXLINKS; i++ ) {
1007        l = &(le_hdlr.links[i]);
1008
1009        /* If the link is active, decrease its interval timer */
1010        if ( l->link_state != LINK_INACTIVE ) {
1011            --l->interval_timer;
1012            if ( l->interval_timer == 0 ) {
1013                /* Automatically increase connection count */
1014                ++(l->conn_count);
1015
1016                /* Automatically adjust to proper unmapped channel */
1017                u8 old_idx = l->channel_idx; /* to revert, if needed.
1018 */
1019                l->channel_idx =
1020                    (l->channel_idx + l->channel_increment) % 37;
1021
1022                /* Apply any connection parameter update if necessary
1023 */
1024                if ( l->update_pending && l->conn_count == \

```



```

1075             le_multi_reset_interval_timer(1);
1076         }
1077     }
1078     /* Every 1.28 seconds , clear any OLD connections. */
1079     /* 312.5us * 4096 */
1080     if( !(clk_n & 0x00000fff) ) {
1081         le_multi_cleanup();
1082     }
1083 }
1084
1085 else if (hop_mode == HOP_AFH) {
1086     if( (last_hop + hop_timeout) == clk_n ) {
1087         do_hop = 1;
1088     }
1089 }
1090
1091 /* Fix linear clock drift deviation */
1092 if( clk_n_next_drift_fix != 0 && clk100ns_offset == 0) {
1093     if( clk_n >= clk_n_next_drift_fix) {
1094         /* Too fast */
1095         if( clk_drift_ppm >= 0) {
1096             clk100ns_offset = 1;
1097         }
1098
1099         /* Too slow */
1100     else {
1101         clk100ns_offset = 6249;
1102     }
1103     clk_n_last_drift_fix = clk_n;
1104     clk_n_next_drift_fix =
1105         clk_n_last_drift_fix + clk_drift_correction;
1106     }
1107 }
1108
1109 /* Negative clock correction */
1110 if( clk100ns_offset > 3124)
1111     clk_n += 2;
1112
1113 T0MR0 = 3124 + clk100ns_offset;
1114 clk100ns_offset = 0;
1115
1116 /* Ack interrupt */
1117 T0IR = TIR_MR0_Interrupt;
1118 }
1119 }
1120
1121 /*
1122 * Function: EINT3_IRQHandler
1123 *
1124 * Also defined in ubertooth.c for TC13BADGE.
1125 */
1126
1127 #ifndef TC13BADGE
1128 void EINT3_IRQHandler() {
1129     /* TODO - check specific source of shared interrupt */
1130     IO2IntClr = PIN_GIO6; /* clear interrupt */
1131     DIO_SSEL_CLR; /* enable SPI */

```

```

1132     cs_trigger = 1;          /* signal trigger */
1133     if (hop_mode == HOP_BLUETOOTH)
1134         dma_discard = 0;
1135 }
1136 #endif /* TC13BADGE */
1137
1138 /*
1139 * Function: msleep
1140 *
1141 * Sleep (busy wait) for 'millis' milliseconds. The 'wait' routines in
1142 * ubertooth.c are matched to the clock setup at boot time and can not
1143 * be used while the board is running at 100MHz.
1144 * msleep:
1145 */
1146 static void msleep(uint32_t millis) {
1147     /* millis -> clkns ticks */
1148     uint32_t stop_at = clkns + millis * 3125 / 1000;
1149     do { } while (clkns < stop_at); /* TODO: handle wrapping */
1150 }
1151
1152 /*
1153 * Function: DMA_IRQHandler
1154 *
1155 * Handler for DMA-related interrupts.
1156 */
1157 void DMA_IRQHandler() {
1158     if (mode == MODE_RX_SYMBOLS
1159         || mode == MODE_SPECAN
1160         || mode == MODE_BT_FOLLOW_LLE
1161         || mode == MODE_BT_MULTIFOLLOW_LLE
1162         || mode == MODE_BT_PROMISC_LLE
1163         || mode == MODE_BT_SLAVE_LLE
1164         || mode == MODE_RX_GENERIC)
1165     {
1166         /* interrupt on channel 0 */
1167         if (DMACIntStat & (1 << 0)) {
1168             if (DMACIntTCStat & (1 << 0)) {
1169                 DMACIntTCClear = (1 << 0);
1170
1171                 if (hop_mode == HOP_BLUETOOTH)
1172                     DIO_SSEL_SET;
1173
1174                 idle_buf_clk100ns = CLK100NS;
1175                 idle_buf_clkn_high = (clkns >> 20) & 0xff;
1176                 idle_buf_channel = channel;
1177
1178                 /* Keep buffer swapping in sync with DMA. */
1179                 volatile uint8_t* tmp = active_rdbuf;
1180                 active_rdbuf = idle_rdbuf;
1181                 idle_rdbuf = tmp;
1182
1183                 ++rx_tc;
1184             }
1185             if (DMACIntErrStat & (1 << 0)) {
1186                 DMACIntErrClr = (1 << 0);
1187                 ++rx_err;
1188             }

```

```

1189         }
1190     }
1191 }
1192
1193 /*
1194 * Function: cc2400_idle
1195 * _____
1196 */
1197 static void cc2400_idle() {
1198     cc2400_strobe(SRFOFF);
1199     while ((cc2400_status() & FS_LOCK)); /* need to wait for unlock? */
1200
1201 #ifdef UBERTOOTHLONE
1202     PAEN_CLR;
1203     HGM_CLR;
1204 #endif
1205
1206     RXLED_CLR;
1207     TXLED_CLR;
1208     USRLED_CLR;
1209
1210     clkn_stop();
1211     dio_ssp_stop();
1212     cs_reset();
1213     rssi_reset();
1214
1215     /* hopping stuff */
1216     hop_mode = HOP_NONE;
1217     do_hop = 0;
1218     channel = 2441;
1219     hop_direct_channel = 0;
1220     hop_timeout = 158;
1221     requested_channel = 0;
1222     saved_request = 0;
1223
1224     /* bulk USB stuff */
1225     idle_buf_clk_high = 0;
1226     idle_buf_clk100ns = 0;
1227     idle_buf_channel = 0;
1228     dma_discard = 0;
1229     status = 0;
1230
1231     /* operation mode */
1232     mode = MODE_IDLE;
1233     requested_mode = MODE_IDLE;
1234     jam_mode = JAM_NONE;
1235     ego_mode = EGO_FOLLOW;
1236
1237     modulation = MOD_BT_BASIC_RATE;
1238
1239     /* specan stuff */
1240     low_freq = 2400;
1241     high_freq = 2483;
1242     rssi_threshold = -30;
1243
1244     target.address = 0;
1245     target.syncword = 0;
1246 }
1247

```

```

1248 /*
1249 * Function: cc2400_rx
1250 *
1251 * Start un-buffered rx.
1252 */
1253 static void cc2400_rx() {
1254     u16 mdmctrl = 0;
1255
1256     if ((modulation == \
1257         MOD_BT_BASIC_RATE) || (modulation == MOD_BT_LOW_ENERGY)) {
1258         if (modulation == MOD_BT_BASIC_RATE) {
1259             mdmctrl = 0x0029; /* 160 kHz frequency deviation */
1260         } else if (modulation == MOD_BT_LOW_ENERGY) {
1261             mdmctrl = 0x0040; /* 250 kHz frequency deviation */
1262         }
1263         cc2400_set(MANAND, 0x7fff);
1264         cc2400_set(LMTST, 0x2b22);
1265         cc2400_set(MDMTST0, 0x134b); /* without PRNG */
1266         cc2400_set(GRMDM, 0x0101); /* un-buffered mode, GFSK */
1267         /* 0 00 00 0 010 00 0 00 0 1 */
1268         /* | | | +-----> CRC off */
1269         /* | | +-----> sync word: 8 MSB bits of SYNC_WORD */
1270         /* | +-----> 2 preamble bytes of 01010101 */
1271         /* | +-----> not packet mode */
1272         /* +-----> un-buffered mode */
1273         cc2400_set(FSDIV, channel - 1); /* 1 MHz IF */
1274         cc2400_set(MDMCTRL, mdmctrl);
1275     }
1276
1277     /* Set up CS register */
1278     cs_threshold_calc_and_set(channel);
1279
1280     clk_n_start();
1281
1282     while (!(cc2400_status() & XOSC16M_STABLE));
1283     cc2400_strobe(SFSON);
1284     while (!(cc2400_status() & FS_LOCK));
1285     cc2400_strobe(SRX);
1286 #ifdef UBERTOOTHTHREE
1287     PAEN_SET;
1288     HGM_SET;
1289 #endif
1290 }
1291
1292 /*
1293 * Function: cc2400_rx_sync
1294 *
1295 * Start un-buffered rx of packets with AA of 'sync'.
1296 * sync: target access address
1297 */
1298 static void cc2400_rx_sync(u32 sync) {
1299     u16 grmdm, mdmctrl;
1300
1301     if (modulation == MOD_BT_BASIC_RATE) {
1302         mdmctrl = 0x0029; /* 160 kHz frequency deviation */
1303         grmdm = 0x0461; /* un-buffered, packet w/ sync word detection */
1304         /* 0 00 00 1 000 11 0 00 0 1 */

```

```

1305     /* | | | | +----> CRC off */
1306     /* | | | +----> sync word: 32 MSB bits of SYNC_WORD */
1307     /* | | +----> 0 preamble bytes of 01010101 */
1308     /* | +----> packet mode */
1309     /* +----> un-buffered mode */
1310     /* +----> sync error bits: 0 */
1311
1312 } else if (modulation == MOD_BT_LOW_ENERGY) {
1313     mdmctrl = 0x0040; /* 250 kHz frequency deviation */
1314     grmdm = 0x0561; /* un-buffered, packet w/ sync word detection */
1315     /* 0 00 00 1 010 11 0 00 0 1 */
1316     /* | | | | +----> CRC off */
1317     /* | | | +----> sync word: 32 MSB bits of SYNC_WORD */
1318     /* | | +----> 2 preamble bytes of 01010101 */
1319     /* | +----> packet mode */
1320     /* +----> un-buffered mode */
1321     /* +----> sync error bits: 0 */
1322
1323 } else {
1324     /* oops */
1325     return;
1326 }
1327
1328 cc2400_set(MANAND, 0x7fff);
1329 cc2400_set(LMTST, 0x2b22);
1330
1331 cc2400_set(MDMTST0, 0x124b);
1332 /* 1 2 4b */
1333 /* 00 0 1 0 0 10 01001011 */
1334 /* | | | | +----> AFC_DELTA = ?? */
1335 /* | | | +----> AFC settling = 4 pairs (8 bit preamble) */
1336 /* | | +----> no AFC adjust on packet */
1337 /* | +----> do not invert data */
1338 /* | +----> TX IF freq 1 0Hz */
1339 /* +----> PRNG off */
1340 /* */
1341 /* ref: CC2400 datasheet page 67 */
1342 /* AFC settling explained page 41/42 */
1343
1344 cc2400_set(GRMDM, grmdm);
1345
1346 cc2400_set(SYNCL, sync & 0xffff);
1347 cc2400_set(SYNCH, (sync >> 16) & 0xffff);
1348
1349 cc2400_set(FSDIV, channel - 1); /* 1 MHz IF */
1350 cc2400_set(MDMCTRL, mdmctrl);
1351
1352 /* Set up CS register */
1353 cs_threshold_calc_and_set(channel);
1354
1355 clk_n_start();
1356
1357 while (!(cc2400_status() & XOSC16M_STABLE));
1358 cc2400_strobe(SFSON);
1359 while (!(cc2400_status() & FS_LOCK));
1360 cc2400_strobe(SRX);
1361 #ifdef UBERTOOTHLONE

```

```

1362     PAEN_SET;
1363     HGMLSET;
1364 #endif
1365 }
1366 /*
1367 * Function: cc2400_tx_sync
1368 */
1369 * Start buffered tx with AA of 'sync'.
1370 * sync: target access address
1371 */
1372 static void cc2400_tx_sync(uint32_t sync) {
1373 #ifdef TX_ENABLE
1374     /* Bluetooth-like modulation */
1375     cc2400_set(MANAND, 0x7fff);
1376     cc2400_set(LMTST, 0x2b22); /* LNA and receive mixers test register */
1377     cc2400_set(MDMTST0, 0x134b); /* no PRNG */
1378
1379     cc2400_set(GRMDM, 0x0c01);
1380     /* 0 00 01 1 000 00 0 00 0 1 */  

1381     /* | | | | +-----> CRC off */  

1382     /* | | | +-----> sync word: 8 MSB bits of SYNC_WORD */  

1383     /* | | +-----> 0 preamble bytes of 01010101 */  

1384     /* | +-----> packet mode */  

1385     /* +-----> buffered mode */
1386
1387     cc2400_set(SYNCL, sync & 0xffff);
1388     cc2400_set(SYNCH, (sync >> 16) & 0xffff);
1389
1390     cc2400_set(FSDIV, channel);
1391     /* amplifier level (-7 dBm, picked from hat) */
1392     cc2400_set(FREND, 0b1011);
1393
1394     if (modulation == MOD_BT_BASIC_RATE) {
1395         cc2400_set(MDMCTRL, 0x0029); /* 160 kHz frequency deviation */
1396     } else if (modulation == MOD_BT_LOW_ENERGY) {
1397         cc2400_set(MDMCTRL, 0x0040); /* 250 kHz frequency deviation */
1398     } else {
1399         /* oops */
1400         return;
1401     }
1402 }
1403
1404 clk_n_start();
1405
1406 while (!(cc2400_status() & XOSC16M_STABLE));
1407 cc2400_strobe(SFS0N);
1408 while (!(cc2400_status() & FS_LOCK));
1409
1410 #ifdef UBERTOOTH_ONE
1411     PAEN_SET;
1412 #endif
1413
1414     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1415     cc2400_strobe(STX);
1416
1417#endif

```

```

1418 }
1419
1420 /* Function: le_transmit
1421 * _____
1422 *
1423 * Transmit a BTLE packet with the specified access address.
1424 * All modulation parameters are set within this function. The data
1425 * should not be pre-whitened, but the CRC should be calculated and
1426 * included in the data length.
1427 * aa:
1428 * len:
1429 * data:
1430 */
1431 void le_transmit(u32 aa, u8 len, u8 *data) {
1432     unsigned i, j;
1433     int bit;
1434     u8 txbuf[64];
1435     u8 tx_len;
1436     u8 byte;
1437     u16 gio_save;
1438
1439     /* first four bytes: AA */
1440     for (i = 0; i < 4; ++i) {
1441         byte = aa & 0xff;
1442         aa >>= 8;
1443         txbuf[i] = 0;
1444         for (j = 0; j < 8; ++j) {
1445             txbuf[i] |= (byte & 1) << (7 - j);
1446             byte >>= 1;
1447         }
1448     }
1449
1450     /* whiten the data and copy it into the txbuf */
1451     int idx = whitening_index[btle_channel_index(channel - 2402)];
1452     for (i = 0; i < len; ++i) {
1453         byte = data[i];
1454         txbuf[i+4] = 0;
1455         for (j = 0; j < 8; ++j) {
1456             bit = (byte & 1) ^ whitening[idx];
1457             idx = (idx + 1) % sizeof(whitening);
1458             byte >>= 1;
1459             txbuf[i+4] |= bit << (7 - j);
1460         }
1461     }
1462
1463     len += 4; /* include the AA in len */
1464
1465     /* Bluetooth-like modulation */
1466     cc2400_set(MANAND, 0x7fff);
1467     /* LNA and receive mixers test register */
1468     cc2400_set(LMTST, 0x2b22);
1469     cc2400_set(MDMTST0, 0x134b); /* no PRNG */
1470
1471     cc2400_set(GRMDM, 0x0c01);
1472     /* 0 00 01 1 000 00 0 00 0 1 */
1473     /* | | | | +-----> CRC off */
1474     /* | | | | +-----> sync word: 8 MSB bits of SYNCWORD */

```

```

1475  /* | +-----> 0 preamble bytes of 01010101 */
1476  /* | +-----> packet mode */
1477  /* +-----> buffered mode */
1478
1479  cc2400_set(FSDIV, channel);
1480  /* amplifier level (-7 dBm, picked from hat) */
1481  cc2400_set(FREND, 0b1011);
1482  cc2400_set(MDMCTRL, 0x0040); /* 250 kHz frequency deviation */
1483  cc2400_set(INT, 0x0014); /* FIFO_THRESHOLD: 20 bytes */
1484
1485  /* sync byte depends on the first transmitted bit of the AA */
1486  if (aa & 1)
1487      cc2400_set(SYNCH, 0xaaaa);
1488  else
1489      cc2400_set(SYNCH, 0x5555);
1490
1491  /* set GIO to FIFO_FULL */
1492  gio_save = cc2400_get(IOCFIG);
1493  cc2400_set(IOCFIG, (GIO_FIFO_FULL << 9) | (gio_save & 0x1ff));
1494
1495  while (!(cc2400_status() & XOSC16M_STABLE));
1496  cc2400_strobe(SFSON);
1497  while (!(cc2400_status() & FS_LOCK));
1498  TXLED_SET;
1499 #ifdef UBERTOOTHLONE
1500  PAEN_SET;
1501#endif
1502  while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1503  cc2400_strobe(STX);
1504
1505  /* put the packet into the FIFO */
1506  for (i = 0; i < len; i += 16) {
1507      while (GIO6); /* wait for the FIFO to drain (FIFO_FULL false) */
1508      tx_len = len - i;
1509      if (tx_len > 16)
1510          tx_len = 16;
1511      cc2400_fifo_write(tx_len, txbuf + i);
1512  }
1513
1514  while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1515  TXLED_CLR;
1516
1517  cc2400_strobe(SRFOFF);
1518  while ((cc2400_status() & FS_LOCK));
1519
1520 #ifdef UBERTOOTHLONE
1521  PAEN_CLR;
1522#endif
1523
1524  /* reset GIO */
1525  cc2400_set(IOCFIG, gio_save);
1526 }
1527
1528 /*
1529 * Function: le_jam
1530 * _____
1531 *
1532 */

```

```

1533 void le_jam( void ) {
1534 #ifdef TX_ENABLE
1535     cc2400_set(MANAND, 0x7fff);
1536     /* LNA and receive mixers test register */
1537     cc2400_set(LMTST, 0x2b22);
1538     cc2400_set(MDMTST0, 0x234b); /* PRNG, 1 MHz offset */
1539
1540     cc2400_set(GRMDM, 0x0c01);
1541     /* 0 00 01 1 000 00 0 00 0 1 */  

1542     /* | | | | +-----> CRC off */  

1543     /* | | | +-----> sync word: 8 MSB bits of SYNC_WORD */  

1544     /* | | +-----> 0 preamble bytes of 01010101 */  

1545     /* | +-----> packet mode */  

1546     /* +-----> buffered mode */
1547
1548     /* amplifier level (-7 dBm, picked from hat) */
1549     cc2400_set(FREND, 0b1011);
1550     cc2400_set(MDMCTRL, 0x0040); /* 250 kHz frequency deviation */
1551
1552     while ( !(cc2400_status() & XOSC16M_STABLE));
1553     cc2400_strobe(SFS0N);
1554     while ( !(cc2400_status() & FS_LOCK));
1555     TXLED_SET;
1556 #ifdef UBERTOOTHTHONE
1557     PAEN_SET;
1558 #endif
1559     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1560     cc2400_strobe(STX);
1561 #endif
1562 }
1563
1564 /*
1565 * Function: hop
1566 *
1567 * Physically change frequency channel based on hop mode and
1568 * hopping parameters.
1569 */
1570 void hop( void ) {
1571     /* TODO - return whether hop happened, or should caller have to keep
1572      * track of this? */
1573     hop_reason = 0;
1574     do_hop = 0;
1575     last_hop = clkn;
1576
1577     /* No hopping, if channel is set correctly, do nothing */
1578     if (hop_mode == HOP_NONE) {
1579         if (cc2400_get(FSDIV) == (channel - 1))
1580             return;
1581     }
1582     /* Slow sweep (100 hops/sec)
1583      * only hop to currently used channels if AFH is enabled
1584      */
1585     else if (hop_mode == HOP_SWEEP) {
1586         do {
1587             channel += 32;
1588             if (channel > 2480)
1589                 channel -= 79;

```

```

1590     } while ( used_channels != 0 && afh_enabled && \
1591       !( afh_map [(channel-2402)/8] & 0x1<<((channel-2402)%8) ) );
1592   }
1593
1594 /* AFH detection
1595 * only hop to currently unused channels
1596 */
1597 else if (hop_mode == HOP_AFH) {
1598   do {
1599     channel += 32;
1600     if (channel > 2480)
1601       channel -= 79;
1602   } while ( used_channels != 79 && \
1603     (afh_map [(channel-2402)/8] & 0x1<<((channel-2402)%8)) );
1604 }
1605
1606 else if (hop_mode == HOP_BLUETOOTH) {
1607   channel = next_hop(clkn);
1608 }
1609
1610 else if (hop_mode == HOP_BTLE) {
1611   channel = btle_next_hop(&le);
1612 }
1613
1614 else if (hop_mode == HOP_BTLE_MULTI) {
1615   /* For re-schedules: scheduler_en == 1 */
1616   /* For hops only: scheduler_en == 0 */
1617   if (le_hdlr.scheduler_en) {
1618     le_multi_scheduler(); /* Changes cur_link */
1619   } else le_hdlr.scheduler_en = 1; /* Re-enable if disabled */
1620
1621   /* Reset the access address in case our link changed */
1622   cc2400_set(SYNC1, le_hdlr.cur_link->sync1);
1623   cc2400_set(SYNC0, le_hdlr.cur_link->sync0);
1624
1625   /* Move on the link's current channel. */
1626   channel = btle_channel_index_to_phys(\ 
1627     btle_afh_channel_index(le_hdlr.cur_link));
1628 }
1629
1630 else if (hop_mode == HOP_DIRECT) {
1631   channel = hop_direct_channel;
1632 }
1633
1634 /* IDLE mode, but leave amp on, so don't call cc2400_idle(). */
1635 cc2400_strobe(SRFOFF);
1636 while ((cc2400_status() & FS_LOCK)); /* need to wait for unlock? */
1637
1638 /* Retune */
1639 if (mode == MODE_TX_SYMBOLS)
1640   cc2400_set(FSDIV, channel);
1641 else
1642   cc2400_set(FSDIV, channel - 1);
1643
1644 /* Update CS register if hopping. */
1645 if (hop_mode > 0) {
1646   cs_threshold_calc_and_set(channel);

```

```

1647     }
1648
1649     /* Wait for lock */
1650     cc2400_strobe(SFSON);
1651     while (!(cc2400_status() & FS_LOCK));
1652
1653     dma_discard = 1;
1654
1655     if (mode == MODE_TX_SYMBOLS)
1656         cc2400_strobe(STX);
1657     else
1658         cc2400_strobe(SRX);
1659 }
1660
1661 /*
1662 * Function: bt_stream_rx
1663 * _____
1664 * Bluetooth packet monitoring.
1665 */
1666 void bt_stream_rx() {
1667
1668     int8_t rssi;
1669     int8_t rssi_at_trigger;
1670
1671     RXLED_CLR;
1672
1673     queue_init();
1674     dio_ssp_init();
1675     dma_init();
1676     dio_ssp_start();
1677
1678     cc2400_rx();
1679
1680     cs_trigger_enable();
1681
1682     while (requested_mode == MODE_RX_SYMBOLS || \
1683           requested_mode == MODE_BT_FOLLOW)
1684     {
1685
1686         RXLED_CLR;
1687
1688         /* Wait for DMA transfer. TODO – need more work on
1689          * RSSI. Should send RSSI indications to host even
1690          * when not transferring data. That would also keep
1691          * the USB stream going. This loop runs 50–80 times
1692          * while waiting for DMA, but RSSI sampling does not
1693          * cover all the symbols in a DMA transfer. Can not do
1694          * RSSI sampling in CS interrupt, but could log time
1695          * at multiple trigger points there. The MAX() below
1696          * helps with statistics in the case that cs_trigger
1697          * happened before the loop started. */
1698         rssi_reset();
1699         rssi_at_trigger = INT8_MIN;
1700         while (!rx_tc) {
1701             rssi = (int8_t)(cc2400_get(RSSI) >> 8);
1702             if (cs_trigger && (rssi_at_trigger == INT8_MIN)) {
1703                 rssi = MAX(rssi, (cs_threshold_cur+54));
1704                 rssi_at_trigger = rssi;

```

```

1705     }
1706     rssi_add(rssi);
1707
1708     handle_usb(clkn);
1709
1710     /* If timer says time to hop, do it. */
1711     if (do_hop) {
1712         hop();
1713     } else {
1714         TXLED_CLR;
1715     }
1716     /* TODO - set per-channel carrier sense threshold.
1717      * Set by firmware or host. */
1718 }
1719
1720 RXLED_SET;
1721
1722 if (rx_err) {
1723     status |= DMA_ERROR;
1724 }
1725
1726 /* Missed a DMA trasfer? */
1727 if (rx_tc > 1)
1728     status |= DMA_OVERFLOW;
1729
1730 if (dma_discard) {
1731     status |= DISCARD;
1732     dma_discard = 0;
1733 }
1734
1735 rssi_iir_update(channel);
1736
1737 /* Set squelch hold if there was either a CS trigger, squelch
1738  * is disabled, or if the current rssi_max is above the same
1739  * threshold. Currently, this is redundant, but allows for
1740  * per-channel or other rssi triggers in the future. */
1741 if (cs_trigger || cs_no_squelch) {
1742     status |= CS_TRIGGER;
1743     cs_trigger = 0;
1744 }
1745
1746 if (rssi_max >= (cs_threshold_cur + 54)) {
1747     status |= RSSL_TRIGGER;
1748 }
1749
1750 enqueue(BR_PACKET, (uint8_t*)idle_rxbuf);
1751
1752 rx_continue:
1753     handle_usb(clkn);
1754     rx_tc = 0;
1755     rx_err = 0;
1756 }
1757
1758 /* This call is a nop so far. Since bt_rx_stream() starts the
1759  * stream, it makes sense that it would stop it. TODO - how
1760  * should setup/teardown be handled? Should every new mode be
1761  * starting from scratch? */
1762 dio_ssp_stop();

```

```

1763     cs_trigger_disable();
1764 }
1765
1766 /*
1767 * Function: reverse8
1768 */
1769 * Reverse the bit order of an 8-bit unsigned number.
1770 * data: 8-bit value to be reversed.
1771 */
1772 static uint8_t reverse8(uint8_t data) {
1773     uint8_t reversed = 0;
1774
1775     for(size_t i=0; i<8; i++)
1776     {
1777         reversed |= ((data >> i) & 0x01) << (7-i);
1778     }
1779
1780     return reversed;
1781 }
1782
1783 /*
1784 * Function: reverse16
1785 */
1786 * Reverse the bit order of a 16-bit unsigned number.
1787 * data: 16-bit value to be reversed.
1788 */
1789 static uint16_t reverse16(uint16_t data) {
1790     uint16_t reversed = 0;
1791
1792     for(size_t i=0; i<16; i++)
1793     {
1794         reversed |= ((data >> i) & 0x01) << (15-i);
1795     }
1796
1797     return reversed;
1798 }
1799
1800 /*
1801 * Function: br_transmit
1802 */
1803 * Transmit a BTBR packet with the specified access code.
1804 * All modulation parameters are set within this function.
1805 */
1806 void br_transmit() {
1807     uint16_t gio_save;
1808
1809     uint32_t clkn_saved = 0;
1810
1811     uint16_t preamble = (target.syncword & 1) == 1 ? 0x5555 : 0xaaaa;
1812     uint8_t trailer = ((target.syncword >> 63) & 1) == 1 ? 0xaa : 0x55;
1813
1814     uint8_t data[16] = {
1815         reverse8((target.syncword >> 0) & 0xFF),
1816         reverse8((target.syncword >> 8) & 0xFF),
1817         reverse8((target.syncword >> 16) & 0xFF),
1818         reverse8((target.syncword >> 24) & 0xFF),
1819         reverse8((target.syncword >> 32) & 0xFF),
1820         reverse8((target.syncword >> 40) & 0xFF),

```

```

1821     reverse8((target.syncword >> 48) & 0xFF),
1822     reverse8((target.syncword >> 56) & 0xFF),
1823     reverse8(trailer),
1824     reverse8(0x77),
1825     reverse8(0x66),
1826     reverse8(0x55),
1827     reverse8(0x44),
1828     reverse8(0x33),
1829     reverse8(0x22),
1830     reverse8(0x11)
1831 };
1832
1833 cc2400_tx_sync(reverse16(preamble));
1834
1835 cc2400_set(INT, 0x0014); /* FIFO_THRESHOLD: 20 bytes */
1836
1837 /* set GIO to FIFO_FULL */
1838 gio_save = cc2400_get(IOCFC);
1839 cc2400_set(IOCFC, (GIO_FIFO_FULL << 9) | (gio_save & 0x1ff));
1840
1841 while (requested_mode == MODE_TX_SYMBOLS)
1842 {
1843     while ((clk_n >> 1) == (clk_n_saved >> 1) || T0TC < 2250) {
1844
1845         /* If timer says time to hop, do it. */
1846         if (do_hop) {
1847             hop();
1848         }
1849     }
1850
1851     clk_n_saved = clk_n;
1852
1853     TXLED_SET;
1854
1855     cc2400_fifo_write(16, data);
1856
1857     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1858     TXLED_CLR;
1859
1860     cc2400_strobe(SRFOFF);
1861     while ((cc2400_status() & FS_LOCK));
1862
1863     while (!(cc2400_status() & XOSC16M_STABLE));
1864     cc2400_strobe(SFS0N);
1865     while (!(cc2400_status() & FS_LOCK));
1866
1867     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
1868     cc2400_strobe(STX);
1869
1870     handle_usb(clk_n);
1871 }
1872
1873 #ifdef UBERTOOTHLONE
1874     PAEN_CLR;
1875 #endif
1876
1877 /* reset GIO */
1878
```

```

1879     cc2400_set (IOCFG, gio_save);
1880 }
1881
1882 /*
1883 * Function: le_set_access_address
1884 *
1885 * Set the access address and SYNCH and SYNCL values for an
1886 * LE link/connection.
1887 * - aa: requested access address.
1888 * - link: target link for address change.
1889 */
1890 static void le_set_access_address(u32 aa, le_state_t *link) {
1891     u32 aa_rev;
1892
1893     link->access_address = aa;
1894     aa_rev = rbit(aa);
1895     link->syncl = aa_rev & 0xffff;
1896     link->synch = aa_rev >> 16;
1897 }
1898
1899 /*
1900 * Function: reset_le
1901 *
1902 * Reset the LE state variable of the target link.
1903 * link: target link to be reset.
1904 */
1905 void reset_le(le_state_t *link) {
1906     le_set_access_address(0x8e89bed6, link); /* Adv AA */
1907     link->crc_init = 0x555555;
1908     link->crc_init_reversed = 0xAAAAAA;
1909     link->crc_verify = 0;
1910     link->last_packet = 0;
1911
1912     link->link_state = LINK_INACTIVE;
1913
1914     link->channel_idx = 0;
1915     link->channel_increment = 0;
1916
1917     for (int i = 0; i < 37; i++) {link->channel_map[i] = 0x00;}
1918     link->num_used_channels = 0;
1919
1920     link->conn_epoch = 0;
1921     link->interval_timer = 0;
1922     link->conn_interval = 0;
1923     link->conn_count = 0;
1924
1925     link->win_size = 0;
1926     link->win_offset = 0;
1927
1928     link->update_pending = 0;
1929     link->update_instant = 0;
1930     link->interval_update = 0;
1931     link->win_size_update = 0;
1932     link->win_offset_update;
1933
1934     link->last_confirmed_event = 0;
1935     link->next_expected_event = 0;
1936     link->linger_time = 0;
1937     link->>window_widening = 0;

```

```

1938 }
1939
1940 /*
1941 * Function: reset_le_promisc
1942 * _____
1943 * Reset the global state for LE promiscuous mode.
1944 * TODO: target a specific link, as in reset_le
1945 */
1946 void reset_le_promisc(void) {
1947     memset(&le_promisc, 0, sizeof(le_promisc));
1948     le_promisc.smallest_hop_interval = 0xffffffff;
1949 }
1950
1951 /*
1952 * Function: le_multi_scheduler
1953 * _____
1954 * Observes all active connections and their priorities and makes a
1955 * decision about which connection to focus on next. The function runs
1956 * at the end of every connection event, as well as when the connection
1957 * handler internal timer expires.
1958 */
1959 void le_multi_scheduler(void) {
1960     /* Disable all timers – we will re-enable them as necessary */
1961     le_hdlr.conn_event_timer_en = 0;
1962     le_hdlr.decision_timer_en = 0;
1963
1964     /* Reset any other necessary stuff */
1965     le_hdlr.cur_anchor_state = ANCHOR_SEARCH;
1966
1967     /* Find the very NEXT connection event across all connections
1968     * (i.e., lowest interval) */
1969     u32 min_interval = 2147483647;
1970     le_state_t *min_link = NULL, *l = NULL;
1971     for (int i=0; i < MAX_LINKS; i++) {
1972         l = &(le_hdlr.links[i]);
1973         /* Check that the link exists */
1974         if (l->link_state != LINK_INACTIVE){
1975             if (l->interval_timer < min_interval) {
1976                 min_interval = l->interval_timer;
1977                 min_link = l;
1978             }
1979         }
1980     }
1981
1982     /* TODO: The logic here may have to change depending on how we
1983     * select the next channel. */
1984
1985     if ( min_link != NULL ) {
1986         /* Plenty of time until the next event! */
1987         if ( min_interval > MIN_ADV_INTERVALS ) {
1988             le_hdlr.cur_link = &(le_hdlr.adv_link);
1989
1990             /* Set the timer for when we need to come back! */
1991             le_hdlr.decision_timer =
1992                 min_link->interval_timer - MIN_ADV_INTERVALS;
1993             le_hdlr.decision_timer_en = 1;
1994     }

```

```

1995     /* Not enough time - just go to the correct connection event when
1996     * the interval timer expires. */
1997     else {
1998         le_hdlr.adv_timer_en = 0;
1999         le_hdlr.scheduler_en = 0;
2000         le_hdlr.cur_link = min_link;
2001         l = le_hdlr.cur_link;
2002
2003         /* Set the connection event timer - once we move to the
2004         * connection, how long are we willing to wait for the first
2005         * frame? It depends... */
2006
2007         /* add the chunk of time before the hop */
2008         le_hdlr.conn_event_timer = l->interval_timer;
2009         /* Adjust for 312.5us interrupts... the longer the better here,
2010         * so let's take ceiling */
2011         le_hdlr.conn_event_timer +=
2012             DIVIDE_CEIL((2*l->window_widening + l->linger_time),3125);
2013
2014         le_hdlr.conn_event_timer += PACKET_OFFSET;
2015
2016         /* Enable the timer */
2017         le_hdlr.conn_event_timer_en = 1;
2018     }
2019 } else {
2020     /* We don't have connections to listen to, listen to advs */
2021     /* No timers necessary... not a care in the world! */
2022     le_hdlr.cur_link = &(le_hdlr.adv_link);
2023 }
2024 }
2025
2026 /*
2027 * Function: bt_generic_le
2028 * _____
2029 * Generic le mode.
2030 * active_mode:
2031 */
2032 void bt_generic_le(u8 active_mode) {
2033     u8 *tmp = NULL;
2034     u8 hold;
2035     int i, j;
2036     int8_t rssi, rssiattrigger;
2037
2038     modulation = MOD_BT_LOW_ENERGY;
2039     mode = active_mode;
2040
2041     reset_le(&le);
2042
2043     /* enable USB interrupts */
2044     ISER0 = ISER0_ISE_USB;
2045
2046     RXLED_CLR;
2047
2048     queue_init();
2049     dio_ssp_init();
2050     dma_init();
2051     dio_ssp_start();
2052     cc2400_rx();

```

```

2053     cs_trigger_enable();
2054
2055     hold = 0;
2056
2057     while (requested_mode == active_mode) {
2058         if (requested_channel != 0) {
2059             cc2400_strobe(SRFOFF);
2060             /* need to wait for unlock? */
2061             while ((cc2400_status() & FS_LOCK));
2062
2063             /* Retune */
2064             cc2400_set(FSDIV, channel - 1);
2065
2066             /* Wait for lock */
2067             cc2400_strobe(SFSON);
2068             while (!(cc2400_status() & FS_LOCK));
2069
2070             /* RX mode */
2071             cc2400_strobe(SRX);
2072
2073             requested_channel = 0;
2074         }
2075
2076         if (do_hop) {
2077             hop();
2078         } else {
2079             TXLED_CLR;
2080         }
2081
2082         RXLED_CLR;
2083
2084         /* Wait for DMA. Meanwhile keep track of RSSI. */
2085         rssi_reset();
2086         rssi_at_trigger = INT8_MIN;
2087         while ((rx_tc == 0) && (rx_err == 0))
2088         {
2089             rssi = (int8_t)(cc2400_get(RSSI) >> 8);
2090             if (cs_trigger && (rssi_at_trigger == INT8_MIN)) {
2091                 rssi = MAX(rssi, (cs_threshold_cur+54));
2092                 rssi_at_trigger = rssi;
2093             }
2094             rssi_add(rssi);
2095         }
2096
2097         if (rx_err) {
2098             status |= DMA_ERROR;
2099         }
2100
2101         /* No DMA transfer? */
2102         if (!rx_tc)
2103             goto rx_continue;
2104
2105         /* Missed a DMA trasfer? */
2106         if (rx_tc > 1)
2107             status |= DMA_OVERFLOW;
2108
2109         rssi_iir_update(channel);
2110

```

```

2111
2112     /* Set squelch hold if there was either a CS trigger , squelch
2113     * is disabled , or if the current rssi_max is above the same
2114     * threshold. Currently , this is redundant , but allows for
2115     * per-channel or other rssi triggers in the future . */
2116     if (cs_trigger || cs_no_squelch) {
2117         status |= CS_TRIGGER;
2118         hold = CS_HOLD_TIME;
2119         cs_trigger = 0;
2120     }
2121
2122     if (rsssi_max >= (cs_threshold_cur + 54)) {
2123         status |= RSSI_TRIGGER;
2124         hold = CS_HOLD_TIME;
2125     }
2126
2127     /* Hold expired? Ignore data. */
2128     if (hold == 0) {
2129         goto rx_continue;
2130     }
2131     hold--;
2132
2133     /* copy the prev unpacked symbols to the front of the buffer */
2134     memcpy(unpacked , unpacked + DMA_SIZE*8 , DMA_SIZE*8);
2135
2136     /* unpack the new packet to the end of the buffer */
2137     for (i = 0; i < DMA_SIZE; ++i) {
2138         /* output one byte for each received symbol (0x00 or 0x01) */
2139         for (j = 0; j < 8; ++j) {
2140             unpacked[DMA_SIZE*8 + i * 8 + j] = (idle_rxbuff[i] & 0x80) >>
2141             7;
2142             idle_rxbuff[i] <= 1;
2143         }
2144     }
2145     int ret = data_cb(unpacked);
2146     if (!ret) break;
2147
2148     rx_continue:
2149     rx_tc = 0;
2150     rx_err = 0;
2151 }
2152
2153     /* disable USB interrupts */
2154 ICER0 = ICER0_ICE_USB;
2155
2156     /* reset the radio completely */
2157 cc2400_idle();
2158 dio_ssp_stop();
2159 cs_trigger_disable();
2160 }
2161
2162 /*
2163 * Function: bt_le_sync
2164 *
2165 * Time-synced version of bt_generic_le. This function waits for DMA,
2166 * processes incoming packets , and enqueues the packets to send to the
2167 * host .

```

```

2168 * - active_mode: mode of operation, used to detect if user wants to
2169 *           change it.
2170 */
2171 void bt_le_sync(u8 active_mode) {
2172     int i;
2173     int8_t rssi;
2174     static int restart_jamming = 0;
2175
2176     modulation = MOD_BT_LOW_ENERGY;
2177     mode = active_mode;
2178
2179     le.link_state = LINK_LISTENING;
2180
2181     /* enable USB interrupts */
2182     ISER0 = ISER0_ISE_USB;
2183
2184     RXLED_CLR;
2185
2186     queue_init();
2187     dio_ssp_init();
2188     dma_init_le();
2189     dio_ssp_start();
2190
2191     /* bit-reversed access address */
2192     cc2400_rx_sync(rbit(le.access_address));
2193
2194     /* Exit if the mode switches for any reason */
2195     while (requested_mode == active_mode) {
2196
2197         /* First, tune to the requested channel, if any */
2198         if (requested_channel != 0) {
2199             cc2400_strobe(SRFOFF);
2200             /* need to wait for unlock? */
2201             while ((cc2400_status() & FS_LOCK));
2202
2203             /* Retune */
2204             cc2400_set(FSDIV, channel - 1);
2205
2206             /* Wait for lock */
2207             cc2400_strobe(SFSON);
2208             while (!(cc2400_status() & FS_LOCK));
2209
2210             /* RX mode */
2211             cc2400_strobe(SRX);
2212
2213             saved_request = requested_channel;
2214             requested_channel = 0;
2215         }
2216
2217         RXLED_CLR;
2218
2219         /* Wait for DMA. Meanwhile keep track of RSSI. */
2220         /* Jose's Notes: DMA will tell us that it has a packet for us! */
2221         rssi_reset();
2222         while ((rx_tc == 0) && (rx_err == 0) && \
2223             (do_hop == 0) && requested_mode == active_mode)
2224             ;
2225

```

```

2226 rssi = (int8_t)(cc2400_get(RSSI) >> 8);
2227 rssi_min = rssi_max = rssi;
2228
2229 if (requested_mode != active_mode) {
2230     goto cleanup;
2231 }
2232
2233 if (rx_err) {
2234     status |= DMA_ERROR;
2235 }
2236
2237 if (do_hop)
2238     goto rx_flush;
2239
2240 /* No DMA transfer? */
2241 if (!rx_tc)
2242     continue;
2243
2244 /* We assume that rx_tc==1 is what brought us here, let's build a
2245 * frame with the info waiting for us in the buffer */
2246 uint32_t packet[48/4+1] = { 0, };
2247 u8 *p = (u8 *)packet;
2248 packet[0] = le.access_address;
2249
2250 const uint32_t *whit =
2251 whitening_word[btle_channel_index(channel-2402)];
2252 for (i = 0; i < 4; i+= 4) {
2253     uint32_t v = rxbufl[i+0] << 24
2254             | rxbufl[i+1] << 16
2255             | rxbufl[i+2] << 8
2256             | rxbufl[i+3] << 0;
2257     packet[i/4+1] = rbit(v) ^ whit[i/4];
2258 }
2259
2260 unsigned len = (p[5] & 0x3f) + 2;
2261 if (len > 39)
2262     goto rx_flush; /* length is bad, throw it all away. */
2263
2264 /* transfer the minimum number of bytes from the CC2400.. this
2265 * allows us enough time to resume RX for subsequent frames
2266 * on the same channel */
2267 unsigned total_transfers = ((len + 3) + 4 - 1) / 4;
2268 if (total_transfers < 11) {
2269     while (DMACC0DestAddr < \
2270         (uint32_t)rxbufl + 4 * total_transfers && rx_err == 0)
2271         ;
2272 } else { /* max transfers? just wait till DMA's done */
2273     while (DMACC0Config & DMACCxConfig_E && rx_err == 0)
2274         ;
2275 }
2276 DIO_SSP_DMACR &= ~SSPDMACR_RXDMAE;
2277
2278 /* strobe SFSON to allow the resync to occur while we process
2279 * the frame */
2280 cc2400_strobe(SFSON);
2281
2282 /* unwhiten the rest of the packet */

```

```

2283     for (i = 4; i < 44; i += 4) {
2284         uint32_t v = rdbuf1[i+0] << 24
2285             | rdbuf1[i+1] << 16
2286             | rdbuf1[i+2] << 8
2287             | rdbuf1[i+3] << 0;
2288         packet[i/4+1] = rbit(v) ^ whit[i/4];
2289     }
2290
2291     if (le.crc_verify) {
2292         u32 calc_crc = btle_crcgen_lut(le.crc_init_reversed, p + 4, len
2293 );
2294         u32 wire_crc = (p[4+len+2] << 16)
2295             | (p[4+len+1] << 8)
2296             | (p[4+len+0] << 0);
2297         if (calc_crc != wire_crc) /* skip packets with a bad CRC */
2298             goto rx_flush;
2299     }
2300
2301     RXLED_SET;
2302
2303     /* Change the state of connection based on packet contents */
2304     packet_cb((uint8_t *)packet);
2305
2306     /* Queue up the frame to send back to our user */
2307     enqueue(LE_PACKET, (uint8_t *)packet);
2308
2309     le.last_packet = CLK100NS;
2310
rx_flush:
2311     /* this might happen twice, but it's safe to do so */
2312     cc2400_strobe(SFSON);
2313
2314     /* flush any excess bytes from the SSP's buffer */
2315     DIO_SSP_DMACR &= ~SSPDMACR_RXDMAE;
2316     while (SSP1SR & SSPSR_RNE) {
2317         u8 tmp = (u8)DIO_SSP_DR;
2318     }
2319
2320     /* timeout - FIXME this is an ugly hack */
2321     u32 now = CLK100NS;
2322     if (now < le.last_packet)
2323         now += 3276800000; /* handle rollover */
2324     if (/* timeout */
2325         ((le.link_state == LINK_CONNECTED || \
2326         le.link_state == LINK_CONN_PENDING) \
2327         && (now - le.last_packet > 50000000)) \
2328         /* jam finished */ \
2329         || (le_jam_count == 1)
2330     ) {
2331         reset_le(&le);
2332         le_jam_count = 0;
2333         TXLED_CLR;
2334
2335     if (jam_mode == JAM_ONCE) {
2336         jam_mode = JAM_NONE;
2337         requested_mode = MODE_IDLE;
2338         goto cleanup;

```

```

2339     }
2340
2341     /* go back to promisc if the connection dies */
2342     if (active_mode == MODE_BT_PROMISCLE)
2343         goto cleanup;
2344
2345     le.link_state = LINK_LISTENING;
2346
2347     cc2400_strobe(SRFOFF);
2348     while ((cc2400_status() & FS_LOCK));
2349
2350     /* Retune */
2351     channel = (saved_request != 0) ? saved_request : 2402;
2352     restart_jamming = 1;
2353 }
2354
2355 cc2400_set(SYNC1, le.sync1);
2356 cc2400_set(SYNC2, le.sync2);
2357
2358 if (do_hop)
2359     hop();
2360
2361 /*      you can jam but you keep turning off the light      */
2362 if (le_jam_count > 0) {
2363     le_jam();
2364     --le_jam_count;
2365 } else {
2366     /* RX mode */
2367     dma_init_le();
2368     dio_ssp_start();
2369
2370     if (restart_jamming) {
2371         cc2400_rx_sync(rbit(le.access_address));
2372         restart_jamming = 0;
2373     } else {
2374         /* wait till we're in FSLOCK before strobing RX */
2375         while (!(cc2400_status() & FS_LOCK));
2376         cc2400_strobe(SRX);
2377     }
2378 }
2379
2380 rx_tc = 0;
2381 rx_err = 0;
2382 } /* end of while (requested_mode==active_mode) */
2383
2384 cleanup:
2385 /* disable USB interrupts */
2386 ICER0 = ICER0_JCE_USB;
2387
2388 /* reset the radio completely */
2389 cc2400_idle();
2390 dio_ssp_stop();
2391 cs_trigger_disable();
2392 }
2393
2394 /*
2395 * Function: bt_multile_sync
2396 *

```

```

2397 * Time-synced version of bt_generic_le for multiple simultaneous
2398 * connections. This function waits for DMA, processes incoming frames,
2399 * and enqueues the frames to send to the host. No "active_mode" is
2400 * provided to the function, as it is assumed that the mode is
2401 * MODE_BT_MULTIFOLLOWLE.
2402 */
2403 void bt_multile_sync() {
2404     int i;
2405     int8_t rssi;
2406     static int restart_jamming = 0;
2407
2408     modulation = MOD_BT_LOW_ENERGY;
2409     mode = MODE_BT_MULTIFOLLOWLE;
2410
2411     /* enable USB interrupts */
2412     ISER0 = ISER0_ISE_USB;
2413
2414     RXLED_CLR;
2415
2416     queue_init();
2417     dio_ssp_init();
2418     dma_init_le();
2419     dio_ssp_start();
2420
2421     /* Tune in to the access address of the current channel */
2422     /* bit-reversed access address */
2423     cc2400_rx_sync(rbit(le_hdlr.cur_link->access_address));
2424
2425     /* Exit if the mode switches for any reason */
2426     while (requested_mode == MODE_BT_MULTIFOLLOWLE) {
2427         RXLED_CLR;
2428
2429         /* Wait for DMA. Meanwhile keep track of RSSI. */
2430         /* DMA will tell us that it has a packet for us! */
2431         rssi_reset();
2432         while ((rx_tc == 0) && (rx_err == 0) && (do_hop == 0) &&
2433             requested_mode == MODE_BT_MULTIFOLLOWLE)
2434             ;
2435
2436         rssi = (int8_t)(cc2400_get(RSSI) >> 8);
2437         rssi_min = rssi_max = rssi;
2438
2439         /* User wants to change the mode.. let's exit. */
2440         if (requested_mode != MODE_BT_MULTIFOLLOWLE) {
2441             goto cleanup;
2442         }
2443
2444         /* Oops! There was an error with DMA */
2445         if (rx_err) {
2446             status |= DMA_ERROR;
2447         }
2448
2449         /* We're done with this channel. Let's move on with our lives. */
2450         if (do_hop)
2451             goto rx_flush;
2452
2453         /* No DMA transfer? */
2454         if (!rx_tc)

```

```

2455     continue; /* Do nothing. */
2456
2457     /* We assume that rx_tc==1 is what brought us here, let's build a
2458     * frame with the info waiting for us in the buffer */
2459     uint32_t packet[48/4+1] = { 0, };
2460     u8 *p = (u8 *)packet;
2461     packet[0] = le_hdlr.cur_link->access_address;
2462
2463     const uint32_t *whit =
2464     whitening_word[btle_channel_index(channel-2402)];
2465     for (i = 0; i < 4; i+= 4) {
2466         uint32_t v = rdbuf1[i+0] << 24
2467             | rdbuf1[i+1] << 16
2468             | rdbuf1[i+2] << 8
2469             | rdbuf1[i+3] << 0;
2470         packet[i/4+1] = rbit(v) ^ whit[i/4];
2471     }
2472
2473     unsigned len = (p[5] & 0x3f) + 2;
2474     if (len > 39)
2475         goto rx_flush; /* length is bad, throw it all away. */
2476
2477     /* transfer the minimum number of bytes from the CC2400 */
2478     /* this allows us enough time to resume RX for subsequent frames
2479     * on the same channel */
2480     unsigned total_transfers = ((len + 3) + 4 - 1) / 4;
2481     if (total_transfers < 11) {
2482         while (DMACC0DestAddr < \
2483             (uint32_t)rdbuf1 + 4 * total_transfers && rx_err == 0)
2484             ;
2485     } else { /* max transfers? just wait till DMA's done */
2486         while (DMACC0Config & DMACCxConfig_E && rx_err == 0)
2487             ;
2488     }
2489     DIO_SSP_DMACR &= ~SSPDMACR_RXDMAE;
2490
2491     /* strobe SFSON to allow the resync to occur while we process
2492     * the frame */
2493     cc2400_strobe(SFSON);
2494
2495     /* unwhiten the rest of the packet */
2496     for (i = 4; i < 44; i += 4) {
2497         uint32_t v = rdbuf1[i+0] << 24
2498             | rdbuf1[i+1] << 16
2499             | rdbuf1[i+2] << 8
2500             | rdbuf1[i+3] << 0;
2501         packet[i/4+1] = rbit(v) ^ whit[i/4];
2502     }
2503
2504     RXLED_SET;
2505
2506     /* Change the state of connection based on packet contents */
2507     connection_multi_follow_cb((u8 *)packet);
2508
2509     /* Oh no! The packet was discarded for some reason. */
2510     /* if (packet == NULL) goto rx_flush; */
2511

```

```

2512 /* Good packet - queue it up to send to host */
2513 enqueue(LE_PACKET, (uint8_t *)packet);
2514
2515 /* If we weren't on an Adv channel, let's reset and enable
2516 * the event timer. */
2517 if (le_hdlr.cur_link != &(le_hdlr.adv_link)) {
2518     le_hdlr.conn_event_timer = TIFS + PACKET_OFFSET;
2519     le_hdlr.conn_event_timer_en = 1;
2520 }
2521
2522 rx_flush:
2523 /* this might happen twice, but it's safe to do so */
2524 cc2400_strobe(SFSON);
2525
2526 /* flush any excess bytes from the SSP's buffer */
2527 DIO_SSP_DMACR &= ~SSPDMACR_RXDMAE;
2528 while (SSP1SR & SSPSR_RNE) {
2529     u8 tmp = (u8)DIO_SSP_DR;
2530 }
2531
2532 if (do_hop) {
2533     hop();
2534 }
2535
2536 dma_init_le();
2537 dio_ssp_start();
2538
2539 /* wait till we're in FSLOCK before strobing RX */
2540 while (!(cc2400_status() & FS_LOCK));
2541 cc2400_strobe(SRX);
2542
2543 rx_tc = 0;
2544 rx_err = 0;
2545 } /* end of while (requested_mode==active_mode) */
2546
2547 cleanup:
2548 /* disable USB interrupts */
2549 ICER0 = ICER0_ICE_USB;
2550
2551 /* reset the radio completely */
2552 cc2400_idle();
2553 dio_ssp_stop();
2554 cs_trigger_disable();
2555 }
2556
2557 /*
2558 * Function: cb_follow_le
2559 * _____
2560 * Legacy LE packet-following function.
2561 * unpacked:
2562 */
2563 int cb_follow_le(char* unpacked) {
2564     int i, j, k;
2565     int idx = whitening_index[btle_channel_index(channel-2402)];
2566
2567     u32 access_address = 0;
2568     for (i = 0; i < 31; ++i) {
2569         access_address >>= 1;

```

```

2570     access_address |= (unpacked[i] << 31);
2571 }
2572
2573 for (i = 31; i < DMA_SIZE * 8 + 32; i++) {
2574     access_address >>= 1;
2575     access_address |= (unpacked[i] << 31);
2576     if (access_address == le.access_address) {
2577         for (j = 0; j < 46; ++j) {
2578             u8 byte = 0;
2579             for (k = 0; k < 8; k++) {
2580                 int offset = k + (j * 8) + i - 31;
2581                 if (offset >= DMA_SIZE*8*2) break;
2582                 int bit = unpacked[offset];
2583                 if (j >= 4) /* unwhiten data bytes */
2584                     bit ^= whitening[idx];
2585                     idx = (idx + 1) % sizeof(whitening);
2586                 }
2587                 byte |= bit << k;
2588             }
2589             idle_rdbuf[j] = byte;
2590         }
2591
2592     /* verify CRC */
2593     if (le.crc_verify) {
2594         int len = (idle_rdbuf[5] & 0x3f) + 2;
2595         u32 calc_crc = btle_crcgen_lut(le.crc_init_reversed, \
2596             (uint8_t *)idle_rdbuf + 4, len);
2597         u32 wire_crc = (idle_rdbuf[4+len+2] << 16) \
2598             | (idle_rdbuf[4+len+1] << 8) \
2599             | idle_rdbuf[4+len+0];
2600         if (calc_crc != wire_crc) /* skip packets with a bad CRC */
2601             break;
2602     }
2603
2604     /* send to PC */
2605     enqueue(LE_PACKET, (uint8_t *)idle_rdbuf);
2606     RXLED_SET;
2607     packet_cb((uint8_t *)idle_rdbuf);
2608
2609     break;
2610 }
2611 }
2612 return 1;
2613 }
2614
2615 /*
2616 * Function: connection_follow_cb
2617 *
2618 * Observes an LE packet and changes link state accordingly
2619 * (e.g., move from LINK_CONNPENDING to LINK_CONNECTED).
2620 * packet: pointer to the packet to be analyzed.
2621 */
2622 void connection_follow_cb(u8 *packet) {
2623     int i;
2624     u32 aa = 0;
2625
2626     u8 *adv_addr = &packet[ADV_ADDRESS_IDX];

```

```

2627 u8 header = packet[HEADER_IDX];
2628 u8 *data_len = &packet[DATA_LEN_IDX];
2629 u8 *data = &packet[DATA_START_IDX];
2630 u8 *crc = &packet[DATA_START_IDX + *data_len];
2631
2632 if (le.link_state == LINK_CONN_PENDING) {
2633
2634 /* We received a packet in the connection pending state, */
2635 /* so now the device *should* be connected */
2636 le.link_state = LINK_CONNECTED;
2637 le.conn_epoch = clkn;
2638 le.interval_timer = le.conn_interval - 1;
2639 le.conn_count = 0;
2640 le.update_pending = 0;
2641
2642 /* hue hue hue */
2643 if (jam_mode != JAM_NONE)
2644     le_jam_count = JAM_COUNT_DEFAULT;
2645
2646 } else if (le.link_state == LINK_CONNECTED) {
2647     u8 llid = header & 0x03;
2648
2649 /* Apply any connection parameter update if necessary */
2650 if (le.update_pending && le.conn_count == le.update_instant) {
2651     /* This is the first packet received in the connection interval
2652      * for which the new parameters apply */
2653     le.conn_epoch = clkn;
2654     le.conn_interval = le.interval_update;
2655     le.interval_timer = le.interval_update - 1;
2656     le.win_size = le.win_size_update;
2657     le.win_offset = le.win_offset_update;
2658     le.update_pending = 0;
2659 }
2660
2661 if (llid == 0x03 && data[0] == 0x00) {
2662     /* This is a CONNECTIONUPDATEREQ. */
2663     /* The host is changing the connection parameters. */
2664     le.win_size_update = packet[7];
2665     le.win_offset_update = packet[8] + ((u16)packet[9] << 8);
2666     le.interval_update = packet[10] + ((u16)packet[11] << 8);
2667     le.update_instant = packet[16] + ((u16)packet[17] << 8);
2668     if (le.update_instant - le.conn_count < 32767)
2669         le.update_pending = 1;
2670 }
2671
2672 } else if (le.link_state == LINK_LISTENING) {
2673     u8 pkt_type = packet[4] & 0x0F;
2674     if (pkt_type == 0x05) {
2675         /* debug stuff !!! */
2676         TXLED_SET;
2677         debug_clk100ns_count = 0;
2678
2679         /* This is a CONNECTREQ */
2680         /* Master is requesting a connection to the slave */
2681         uint16_t conn_interval;
2682
2683         /* ignore packets with incorrect length */

```

```

2684     if (*data_len != 34)
2685         return;
2686
2687     /* conn_interval must be [7.5 ms, 4.0s] in units of 1.25 ms */
2688     conn_interval = (packet[29] << 8) | packet[28];
2689     if (conn_interval < 6 || conn_interval > 3200)
2690         return;
2691
2692     /* if we have a target , see if InitA or AdvA matches */
2693     if (le.target_set &&
2694         /* Target address doesn't match Initiator. */
2695         memcmp(le.target, &packet[6], 6) &&
2696         /* Target address doesn't match Advertiser. */
2697         memcmp(le.target, &packet[12], 6)) {
2698             return;
2699         }
2700
2701     /* We are now PENDING, since the first hop has not occurred */
2702     le.link_state = LINK_CONN_PENDING;
2703     /* we will drop many packets if we attempt to filter by CRC */
2704     le.crc_verify = 0;
2705
2706     for (i = 0; i < 4; ++i)
2707         aa |= packet[18+i] << (i*8);
2708     /* We now have an access address */
2709     le.set_access_address(aa, &le);
2710     le.crc_init = (packet[CRC_INIT+2] << 16)
2711         | (packet[CRC_INIT+1] << 8)
2712         | packet[CRC_INIT+0];
2713     le.crc_init_reversed = rbit(le.crc_init);
2714     le.win_size = packet[WIN_SIZE];
2715     le.win_offset = packet[WIN_OFFSET];
2716     le.conn_interval = (packet[CONN_INTERVAL+1] << 8)
2717         | packet[CONN_INTERVAL+0];
2718     le.channel_increment = packet[CHANNEL_INC] & 0x1f;
2719     le.channel_idx = le.channel_increment;
2720
2721     /* Hop to the initial channel immediately... */
2722     /* We'll wait for the first packet there */
2723     do_hop = 1;
2724 }
2725 }
2726 }
2727
2728 */
2729 * Function: connection_multi_follow_cb
2730 *
2731 * Observes an LE packet and changes link state accordingly
2732 * (e.g., move from LINK_CONN_PENDING to LINK_CONNECTED). Used when
2733 * following multiple connections, the "active" connection is assumed
2734 * to be le_hdlr.cur_link.
2735 * packet: pointer to the packet to be analyzed.
2736 */
2737 void connection_multi_follow_cb(u8 *packet) {
2738     int i;
2739     u32 aa = 0;
2740

```

```

2741 u8 *adv_addr = &packet[ADV_ADDRESS_IDX];
2742 u8 header = packet[HEADER_IDX];
2743 u8 *data_len = &packet[DATA_LEN_IDX];
2744 u8 *data = &packet[DATA_START_IDX];
2745 u8 *crc = &packet[DATA_START_IDX + *data_len];
2746
2747 /* Our working variable for the current link */
2748 le_state_t *l = NULL;
2749
2750 /* Are we dealing with a connection, or just an adv packet? */
2751 if (le_hdlr.cur_link==&(le_hdlr.adv_link)){
2752     /* If we SEE a connection request for a target of interest,
2753     * create a new connection */
2754     u8 pkt_type = packet[4] & 0x0F;
2755     if (pkt_type == 0x05) {
2756         /* This is a CONNECTREQ */
2757         /* Master is requesting a connection to the slave */
2758         uint16_t conn_interval;
2759
2760         /* ignore packets with incorrect length */
2761         if (*data_len != 34)
2762             return;
2763
2764         /* conn interval must be [7.5 ms, 4.0s] in units of 1.25 ms */
2765         conn_interval = (packet[29] << 8) | packet[28];
2766         if (conn_interval < 6 || conn_interval > 3200)
2767             return;
2768
2769         /* TODO: Does the connection req match one of the targets? */
2770
2771         /* Find some room for our new connection */
2772         l = NULL;
2773         for (int i = 0; i < MAX_LINKS; i++){
2774             if (le_hdlr.links[i].link_state == LINK_INACTIVE) {
2775                 l = &le_hdlr.links[i];
2776                 break;
2777             }
2778         }
2779
2780         /* We don't have room for the connection! */
2781         if (l == NULL) return;
2782
2783         /* We are now PENDING, since the first hop has not occurred */
2784         l->link_state = LINK_CONN_PENDING;
2785         /* we will drop many packets if we attempt to filter by CRC */
2786         l->crc_verify = 0;
2787         for (i = 0; i < 4; ++i)
2788             aa |= packet[18+i] << (i*8);
2789         le_set_access_address(aa, l); /* We now have an AA */
2790
2791         l->crc_init = (packet[CRC_INIT+2] << 16)
2792             | (packet[CRC_INIT+1] << 8)
2793             | packet[CRC_INIT+0];
2794         l->crc_init_reversed = rbit(l->crc_init);
2795         l->win_size = packet[WIN_SIZE] * 12500;
2796         l->win_offset =
2797             (packet[WIN_OFFSET + 1] << 8 | packet[WIN_OFFSET]) * 12500;

```

```

2798     l->conn_interval = conn_interval * 12500;
2799     l->channel_increment = packet[CHANNEL_INC] & 0x1f;
2800     l->channel_idx = 0;
2801
2802     l->num_used_channels = 0;
2803     for (int i = 0; i < 37; i++){
2804         l->channel_map[i] =
2805             (packet[CHANNEL_MAP + i / 8] >> (i % 8)) & 0x01;
2806         if (l->channel_map[i] != 0x00) l->num_used_channels++;
2807     }
2808
2809     u16 sca = 0;
2810     switch ((packet[CHANNEL_INC] >> 5) & 0x07) {
2811         case 0: sca = 500; break;
2812         case 1: sca = 250; break;
2813         case 2: sca = 150; break;
2814         case 3: sca = 100; break;
2815         case 4: sca = 75; break;
2816         case 5: sca = 50; break;
2817         case 6: sca = 30; break;
2818         case 7: sca = 20; break;
2819     }
2820     l->sca = sca;
2821 /* At the expiration of the timer, we move to count = 0. */
2822     l->conn_count = -1;
2823
2824     /* Set the connection's first interval timer
2825      * (when is the window opening?) */
2826     l->linger_time = l->win_size;
2827     l->last_confirmed_event = CLK100NS;
2828     l->next_expected_event =
2829         l->last_confirmed_event + 12500 + l->win_offset;
2830     if (l->next_expected_event >= CLK100NS_MAX){
2831         l->next_expected_event = l->next_expected_event -
2832             CLK100NS_MAX;
2833     }
2834     le_multi_reset_interval_timer(l);
2835
2836     TXLED_SET;
2837
2838     /* Run the scheduler immediately */
2839     do_hop = 1;
2840     hop_reason |= F_INTERVAL;
2841
2842     /* So it's not a connection request... */
2843     /* Is the advertisement packet an adv for one of our targets? */
2844     else {
2845         /* if we have a target, see if the address matches it. */
2846         if (le.target_set && !memcmp(le.target, &packet[6], 6)) {
2847             /* Target matches our set target */
2848             le_hdlr.cur_adv_state = ADV_CANDIDATE;
2849             le_hdlr.adv_timer = 1;
2850             le_hdlr.adv_timer_en = 1;
2851         }
2852     }
2853 }
```

```

2854
2855     /* Grab the connection , make it easier to work with. */
2856     l = le_hdlr.cur_link;
2857     /* TODO: First , let's verify that the packet is valid
2858     * ( if requested!) */
2859     if (l->crc_verify) {
2860         /*
2861     }
2862
2863     /* We received a conn req and are waiting to begin hopping */
2864     if (l->link_state == LINK_CONN_PENDING) {
2865         /* We received a packet in the connection pending state , */
2866         /* so now the device *should* be connected */
2867         l->link_state = LINK_CONNECTED;
2868         l->conn_epoch = CLK100NS;
2869         l->update_pending = 0;
2870
2871         /* Set the interval timer */
2872         if ( le_hdlr.conn_event_timer < l->window_widening ) {
2873             l->linger_time = 0;
2874         } else if ( le_hdlr.conn_event_timer - l->window_widening < \
2875             l->win_size ) {
2876             l->linger_time =
2877             l->win_size - (le_hdlr.conn_event_timer-l->window_widening);
2878         } else {
2879             l->linger_time = l->win_size;
2880         }
2881         l->last_confirmed_event = l->conn_epoch;
2882         l->next_expected_event =
2883         l->conn_epoch + l->conn_interval - l->linger_time;
2884         l->next_expected_event = l->next_expected_event - PACKET_OFFSET
2885     ;
2886         if (l->next_expected_event >= CLK100NS_MAX){
2887             l->next_expected_event = l->next_expected_event -
2888             CLK100NS_MAX;
2889         }
2890         le_multi_reset_interval_timer(l);
2891     }
2892     /* We are already connected and hopping */
2893     else if (l->link_state == LINK_CONNECTED) {
2894         u8 llid = header & 0x03;
2895
2896         /* LL_CONNECTION_UPDATE_REQ */
2897         if (llid == 0x03 && data[0] == 0x00) {
2898             l->win_size_update = packet[7] * 12500;
2899             l->win_offset_update =
2900             (packet[8] + ((u16)packet[9] << 8)) * 12500;
2901             l->interval_update =
2902             (packet[10] + ((u16)packet[11] << 8)) * 12500;
2903             l->update_instant = packet[16] + ((u16)packet[17] << 8);
2904             if (l->update_instant - l->conn_count < 32767)
2905                 l->update_pending = 1;
2906
2907         /* LL_CHANNEL_MAP_REQ */
2908         else if (llid == 0x03 && data[0] == 0x01) {

```

```

2909     l->num_used_channels_update = 0;
2910     for (int i = 0; i < 37; i++){
2911         l->channel_map_update[i] =
2912             (packet[7 + i/8] >> (i % 8)) & 0x01;
2913         if (l->channel_map_update[i] != 0x00)
2914             l->num_used_channels_update++;
2915     }
2916     l->update_instant = packet[12] + ((u16)packet[13] << 8);
2917     if (l->update_instant - l->conn_count < 32767)
2918         l->map_update_pending = 1;
2919 }
2920
2921 /* LL_TERMINATE_IND */
2922 else if (llid == 0x03 && data[0] == 0x02) {
2923     /* Connection ended.. release the link. */
2924     reset_le(1);
2925 };
2926
2927 /* For all DATA CHANNEL frames , see if any is an anchor */
2928 /* is the packet empty? */
2929 int isEmpty = (*data_len == 0) ? 1 : 0;
2930 /* is the MD bit set? */
2931 int isMDClear = (header & 0x10) ? 0 : 1;
2932
2933 if(le_hdlr.cur_anchor_state == ANCHORSEARCH) {
2934     /* Looking for the first EMPTY packet with MD=0 */
2935     if (isEmpty && isMDClear) {
2936         /* Found it! */
2937         le_hdlr.cur_anchor_state = ANCHOR_CANDIDATE;
2938         le_hdlr.candidate_anchor = CLK100NS;
2939     }
2940 } else if (le_hdlr.cur_anchor_state == ANCHOR_CANDIDATE) {
2941     /* Looking for the second EMPTY packet with MD=0 */
2942     if (isEmpty && isMDClear) {
2943         /* Found the second - let's set the anchor point,
2944         * and calibrate our timer */
2945         l->linger_time = 0;
2946         l->last_confirmed_event = le_hdlr.candidate_anchor;
2947         l->next_expected_event =
2948             l->last_confirmed_event + l->conn_interval;
2949         l->next_expected_event =
2950             l->next_expected_event - PACKET_OFFSET;
2951         le_multi_reset_interval_timer(1);
2952     }
2953     le_hdlr.cur_anchor_state = ANCHORSEARCH;
2954 }
2955 }
2956 if (l != NULL) {
2957     l->last_packet = CLK100NS;
2958 }
2959 }
2960 }
2961
2962 /*
2963 * Function: bt_follow_le
2964 */

```

```

2965 * Setup function to follow a single LE connection.
2966 */
2967 void bt_follow_le() {
2968     /* The current requested_mode is MODE_BT_FOLLOWLE */
2969     reset_le(&le);
2970     packet_cb = connection_follow_cb;
2971
2972     bt_le_sync(MODE_BT_FOLLOWLE);
2973
2974     mode = MODE_IDLE;
2975 }
2976
2977 /*
2978 * Function: bt_multi_follow_le
2979 *
2980 * Setup function to follow multiple LE connections simultaneously.
2981 */
2982 void bt_multi_follow_le() {
2983
2984     /* Reset all of our link states */
2985     for (int i=0; i < MAX_LINKS; i++) {
2986         reset_le(&(le_hdlr.links[i]));
2987     }
2988
2989     /* The first place we should start is on an advertisement channel */
2990     le_hdlr.cur_link = &(le_hdlr.adv_link);
2991
2992     /* Start pulling packets out of the air */
2993     bt_multi_le_sync();
2994
2995     /* All done - go IDLE */
2996     mode = MODE_IDLE;
2997 }
2998
2999 /***** Functions for PROMISCUOUS MODE *****/
3000 /*
3001 * Function: le_promisc_state
3002 *
3003 * Issue state change message.
3004 * type:
3005 * data:
3006 * len:
3007 */
3008 void le_promisc_state(u8 type, void *data, unsigned len) {
3009     u8 buf[50] = { 0, };
3010     if (len > 49)
3011         len = 49;
3012
3013     buf[0] = type;
3014     memcpy(&buf[1], data, len);
3015     enqueue(LE_PROMISC, (uint8_t *)buf);
3016 }
3017
3018 /*
3019 * Function: promisc_recover_hop_increment
3020 *
3021 * Determine the LE hop increment parameter while in promiscuous mode.
3022 * packet: pointer to the packet to be analyzed.

```

```

3023 */
3024 void promisc_recover_hop_increment(u8 *packet) {
3025     static u32 first_ts = 0;
3026     if (channel == 2404) {
3027         first_ts = CLK100NS;
3028         hop_direct_channel = 2406;
3029         do_hop = 1;
3030     } else if (channel == 2406) {
3031         u32 second_ts = CLK100NS;
3032         if (second_ts < first_ts)
3033             second_ts += 3276800000; /* handle rollover */
3034         /* # of channels hopped between prev and current timestamp. */
3035         u32 channels_hopped = DIVIDE_ROUND(second_ts - first_ts,
3036                                             le.conn_interval * LE_BASECLK);
3037         if (channels_hopped < 37) {
3038             /* Get the hop increment based on the # of channels hopped. */
3039             le.channel_increment = hop_interval_lut[channels_hopped];
3040             le.interval_timer = le.conn_interval / 2;
3041             le.conn_count = 0;
3042             le.conn_epoch = 0;
3043             do_hop = 0;
3044             /* Move on to regular connection following. */
3045             le.channel_idx = (1 + le.channel_increment) % 37;
3046             le.link_state = LINK_CONNECTED;
3047             le.crc_verify = 0;
3048             hop_mode = HOP_BTLE;
3049             packet_cb = connection_follow_cb;
3050             le_promisc_state(3, &le.channel_increment, 1);
3051
3052             if (jam_mode != JAM_NONE)
3053                 le.jam_count = JAM_COUNT_DEFAULT;
3054
3055             return;
3056         }
3057         hop_direct_channel = 2404;
3058         do_hop = 1;
3059     }
3060     else {
3061         hop_direct_channel = 2404;
3062         do_hop = 1;
3063     }
3064 }
3065
3066 /*
3067 * Function: promisc_recover_hop_interval
3068 *
3069 * Determine the LE hop interval parameter while in promiscuous mode.
3070 * packet: pointer to the packet to be analyzed.
3071 */
3072 void promisc_recover_hop_interval(u8 *packet) {
3073     static u32 prev_clk = 0;
3074
3075     u32 cur_clk = CLK100NS;
3076     if (cur_clk < prev_clk)
3077         cur_clk += 3267800000; /* handle rollover */
3078     u32 clk_diff = cur_clk - prev_clk;
3079     u16 obsv_hop_interval; /* observed hop interval */

```

```

3080
3081 /* probably consecutive data packets on the same channel */
3082 if (clk_diff < 2 * LE_BASECLK)
3083     return;
3084
3085 if (clk_diff < le_promisc.smallest_hop_interval)
3086     le_promisc.smallest_hop_interval = clk_diff;
3087
3088 obsv_hop_interval =
3089     DIVIDEROUND(le_promisc.smallest_hop_interval, 37 * LE_BASECLK);
3090
3091 if (le.conn_interval == obsv_hop_interval) {
3092     /* 5 consecutive hop intervals: consider it legit and move on */
3093     ++le_promisc.consec_intervals;
3094     if (le_promisc.consec_intervals == 5) {
3095         packet_cb = promisc_recover_hop_increment;
3096         hop_direct_channel = 2404;
3097         hop_mode = HOP_DIRECT;
3098         do_hop = 1;
3099         le_promisc_state(2, &le.conn_interval, 2);
3100     }
3101 } else {
3102     le.conn_interval = obsv_hop_interval;
3103     le_promisc.consec_intervals = 0;
3104 }
3105
3106 prev_clk = cur_clk;
3107 }
3108
3109 /*
3110 * Function: promisc_follow_cb
3111 * _____
3112 *
3113 * packet: pointer to the packet to be analyzed.
3114 */
3115 void promisc_follow_cb(u8 *packet) {
3116     int i;
3117
3118     /* get the CRCInit */
3119     if (!le.crc_verify && packet[4] == 0x01 && packet[5] == 0x00) {
3120         u32 crc = (packet[8] << 16) | (packet[7] << 8) | packet[6];
3121
3122         le.crc_init = bt_le_reverse_crc(crc, packet + 4, 2);
3123         le.crc_init_reversed = 0;
3124         for (i = 0; i < 24; ++i)
3125             le.crc_init_reversed |= ((le.crc_init >> i) & 1) << (23 - i);
3126
3127         le.crc_verify = 1;
3128         packet_cb = promisc_recover_hop_interval;
3129         le_promisc_state(1, &le.crc_init, 3);
3130     }
3131 }
3132
3133 /*
3134 * Function: see_aa
3135 * _____
3136 * In promiscuous mode, when an access address is seen on the wire,
3137 * add it to the running list of AA candidates.

```

```

3138 * aa: access address observed on the wire.
3139 */
3140 void see_aa(u32 aa) {
3141     int i, max = -1, killme = -1;
3142     for (i = 0; i < AA_LIST_SIZE; ++i)
3143         if (le_promisc.active_aa[i].aa == aa) {
3144             ++le_promisc.active_aa[i].count;
3145             return;
3146         }
3147
3148 /* evict someone */
3149 for (i = 0; i < AA_LIST_SIZE; ++i)
3150     if (le_promisc.active_aa[i].count < max || max < 0) {
3151         killme = i;
3152         max = le_promisc.active_aa[i].count;
3153     }
3154
3155 le_promisc.active_aa[killme].aa = aa;
3156 le_promisc.active_aa[killme].count = 1;
3157 }
3158
3159 /*
3160 * Function: cb_le_promisc
3161 * _____
3162 * LE promiscuous mode.
3163 * unpacked:
3164 */
3165 int cb_le_promisc(char *unpacked) {
3166     int i, j, k;
3167     int idx;
3168
3169 /* empty data PDU: 01 00 */
3170 char desired[4][16] = {
3171     { 1, 0, 0, 0, 0, 0, 0, 0,
3172       0, 0, 0, 0, 0, 0, 0, 0, },
3173     { 1, 0, 0, 1, 0, 0, 0, 0,
3174       0, 0, 0, 0, 0, 0, 0, 0, },
3175     { 1, 0, 1, 0, 0, 0, 0, 0,
3176       0, 0, 0, 0, 0, 0, 0, 0, },
3177     { 1, 0, 1, 1, 0, 0, 0, 0,
3178       0, 0, 0, 0, 0, 0, 0, 0, },
3179 };
3180
3181 for (i = 0; i < 4; ++i) {
3182     idx = whitening_index[btle_channel_index(channel - 2402)];
3183
3184     /* whiten the desired data */
3185     for (j = 0; j < (int)sizeof(desired[i]); ++j) {
3186         desired[i][j] ^= whitening[idx];
3187         idx = (idx + 1) % sizeof(whitening);
3188     }
3189 }
3190
3191 /* then look for that bitsream in our receive buffer */
3192 for (i = 32; i < (DMA_SIZE*8*2 - 32 - 16); i++) {
3193     int ok[4] = { 1, 1, 1, 1 };
3194     int matching = -1;

```

```

3195
3196     for (j = 0; j < 4; ++j) {
3197         for (k = 0; k < (int)sizeof(desired[j]); ++k) {
3198             if (unpacked[i+k] != desired[j][k]) {
3199                 ok[j] = 0;
3200                 break;
3201             }
3202         }
3203     }
3204
3205     /* see if any match */
3206     for (j = 0; j < 4; ++j) {
3207         if (ok[j]) {
3208             matching = j;
3209             break;
3210         }
3211     }
3212
3213     /* skip if no match */
3214     if (matching < 0)
3215         continue;
3216
3217     /* found a match! unwhiten it and send it home */
3218     idx = whitening_index[btle_channel_index(channel - 2402)];
3219     for (j = 0; j < 4+3+3; ++j) {
3220         u8 byte = 0;
3221         for (k = 0; k < 8; k++) {
3222             int offset = k + (j * 8) + i - 32;
3223             if (offset >= DMA_SIZE*8*2) break;
3224             int bit = unpacked[offset];
3225             if (j >= 4) { /* unwhiten data bytes */
3226                 bit ^= whitening[idx];
3227                 idx = (idx + 1) % sizeof(whitening);
3228             }
3229             byte |= bit << k;
3230         }
3231         idle_rdbuf[j] = byte;
3232     }
3233
3234     u32 aa = (idle_rdbuf[3] << 24) |
3235         (idle_rdbuf[2] << 16) |
3236         (idle_rdbuf[1] << 8) |
3237         (idle_rdbuf[0]);
3238     see_aa(aa);
3239
3240     enqueue(LE_PACKET, (uint8_t*)idle_rdbuf);
3241
3242 }
3243
3244 /* once we see an AA 5 times, start following it */
3245 for (i = 0; i < AA_LIST_SIZE; ++i) {
3246     if (le_promisc.active_aa[i].count > 3) {
3247         le_set_access_address(le_promisc.active_aa[i].aa, &le);
3248         data_cb = cb_follow_le;
3249         packet_cb = promisc_follow_cb;
3250         le.crc_verify = 0;
3251         le_promisc_state(0, &le.access_address, 4);

```

```

3252     /* quit using the old stuff and switch to sync mode */
3253     return 0;
3254 }
3255 }
3256
3257     return 1;
3258 }
3259
3260 /*
3261 * Function: bt_promisc_le
3262 *
3263 * Grab LE packets out of the air while in promiscuous mode.
3264 */
3265 void bt_promisc_le() {
3266     while (requested_mode == MODE_BT_PROMISCLE) {
3267         reset_le_promisc();
3268
3269         /* jump to a random data channel and turn up the squelch */
3270         if ((channel & 1) == 1)
3271             channel = 2440;
3272
3273         /* if the PC hasn't given us AA, determine by listening */
3274         if (!le.target_set) {
3275             /* cs_threshold_req = -80; */
3276             cs_threshold_calc_and_set(channel);
3277             data_cb = cb_le_promisc;
3278             bt_generic_le(MODE_BT_PROMISCLE);
3279         }
3280
3281         /* could have got mode change in middle of above */
3282         if (requested_mode != MODE_BT_PROMISCLE)
3283             break;
3284
3285         le_promisc_state(0, &le.access_address, 4);
3286         packet_cb = promisc_follow_cb;
3287         le.crc_verify = 0;
3288         bt_le_sync(MODE_BT_PROMISCLE);
3289     }
3290 }
3291 **** MISC ****
3292 /*
3293 * Function: bt_slave_le
3294 *
3295 */
3296 void bt_slave_le(void) {
3297     u32 calc_crc;
3298     int i;
3299
3300     u8 adv_ind[] = {
3301         /* LL header */
3302         0x00, 0x09,
3303
3304         /* advertising address */
3305         0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
3306
3307         /* advertising data */
3308         0x02, 0x01, 0x05,
3309

```

```

3310
3311     /* CRC ( calc ) */
3312     0xff, 0xff, 0xff,
3313 };
3314
3315 u8 adv_ind_len = sizeof(adv_ind) - 3;
3316
3317 /* copy the user-specified mac address */
3318 for (i = 0; i < 6; ++i)
3319     adv_ind[i+2] = slave_mac_address[5-i];
3320
3321 calc_crc = btle_calc_crc(le.crc_init_reversed, adv_ind, adv_ind_len);
3322 adv_ind[adv_ind_len+0] = (calc_crc >> 0) & 0xff;
3323 adv_ind[adv_ind_len+1] = (calc_crc >> 8) & 0xff;
3324 adv_ind[adv_ind_len+2] = (calc_crc >> 16) & 0xff;
3325
3326 clkn_start();
3327
3328 /* spam advertising packets */
3329 while (requested_mode == MODE_BT_SLAVE_LE) {
3330     ICER0 = ICER0_ICE_USB;
3331     ICER0 = ICER0_ICE_DMA;
3332     le_transmit(0x8e89bed6, adv_ind_len+3, adv_ind);
3333     ISER0 = ISER0_ISE_USB;
3334     ISER0 = ISER0_ISE_DMA;
3335     msleep(100);
3336 }
3337 }
3338
3339 /*
3340 * Function: rx_generic_sync
3341 *
3342 */
3343 */
3344 void rx_generic_sync(void) {
3345     int i, j;
3346     u8 len = 32;
3347     u8 buf[len+4];
3348     u16 reg_val;
3349
3350     /* Put syncword at start of buffer
3351      * DGS: fix this later, we don't know number of syncword bytes, etc
3352      */
3353     reg_val = cc2400_get(SYNCH);
3354     buf[0] = (reg_val >> 8) & 0xFF;
3355     buf[1] = reg_val & 0xFF;
3356     reg_val = cc2400_get(SYNCL);
3357     buf[2] = (reg_val >> 8) & 0xFF;
3358     buf[3] = reg_val & 0xFF;
3359
3360     queue_init();
3361     clkn_start();
3362
3363     while (!(cc2400_status() & XOSC16M_STABLE));
3364     cc2400_strobe(SFS0N);
3365     while (!(cc2400_status() & FS_LOCK));
3366     RXLED_SET;

```

```

3367 #ifdef UBERTOOTHLONE
3368     PAEN_SET;
3369     HGM_SET;
3370 #endif
3371     while (1) {
3372         while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
3373         cc2400_strobe(SRX);
3374         USRLED_CLR;
3375         while (!(cc2400_status() & SYNC_RECEIVED));
3376         USRLED_SET;
3377
3378         cc2400_fifo_read(len, buf+4);
3379         enqueue(BR_PACKET, buf);
3380         handle_usb(clkn);
3381     }
3382 }
3383
3384 /*
3385 * Function: rx_generic
3386 * _____
3387 *
3388 */
3389 void rx_generic(void) {
3390     /* Check for packet mode */
3391     if (cc2400_get(GRMDM) && 0x0400) {
3392         rx_generic_sync();
3393     } else {
3394         modulation == MOD_NONE;
3395         bt_stream_rx();
3396     }
3397 }
3398
3399 /*
3400 * Function: tx_generic
3401 * _____
3402 *
3403 */
3404 void tx_generic(void) {
3405     u32 i;
3406     u16 synch, syncl;
3407     u8 tx_len;
3408     u8 prev_mode = mode;
3409
3410     mode = MODE_TX_GENERIC;
3411
3412     /* Save existing syncword */
3413     synch = cc2400_get(SYNCH);
3414     syncl = cc2400_get(SYNCL);
3415
3416     cc2400_set(SYNCH, tx_pkt.synch);
3417     cc2400_set(SYNCL, tx_pkt.syncl);
3418     cc2400_set(MDMCTRL, 0x0057);
3419     cc2400_set(MDMTST0, 0x134b);
3420     cc2400_set(GRMDM, 0x0f61);
3421     cc2400_set(FSDIV, tx_pkt.channel);
3422     cc2400_set(FREND, tx_pkt.pa_level);
3423
3424     while (!(cc2400_status() & XOSC16M_STABLE));

```

```

3425     cc2400_strobe(SFSON);
3426     while (!(cc2400_status() & FS_LOCK));
3427     TXLED_SET;
3428 #ifdef UBERTOOTHLONE
3429     PAEN_SET;
3430 #endif
3431     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
3432
3433     cc2400_fifo_write(tx_pkt.length, tx_pkt.data);
3434     cc2400_strobe(STX);
3435
3436     while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
3437     TXLED_CLR;
3438
3439     cc2400_strobe(SRFOFF);
3440     while ((cc2400_status() & FS_LOCK));
3441 #ifdef UBERTOOTHLONE
3442     PAEN_CLR;
3443 #endif
3444
3445 /* Restore state */
3446 cc2400_set(SYNCH, synch);
3447 cc2400_set(_SYNCL, sync1);
3448 requested_mode = prev_mode;
3449 }
3450
3451 /***** Spectrum analyzers *****/
3452 /*
3453 * Function: specan
3454 * _____
3455 * Run spectrum analyzer.
3456 */
3457 void specan(void) {
3458     u16 f;
3459     u8 i = 0;
3460     u8 buf[DMA_SIZE];
3461
3462     RXLED_SET;
3463
3464     queue_init();
3465     clk_n_start();
3466 #ifdef UBERTOOTHLONE
3467     PAEN_SET;
3468     /*HGM_SET; */
3469 #endif
3470     cc2400_set(LMTST, 0x2b22);
3471     cc2400_set(MDMTST0, 0x134b); /* without PRNG */
3472     cc2400_set(GRMDM, 0x0101); /* un-buffered mode, GFSK */
3473     cc2400_set(MDMCTRL, 0x0029); /* 160 kHz frequency deviation */
3474     /*FIXME maybe set RSSI.RSSI_FILT */
3475     while (!(cc2400_status() & XOSC16M_STABLE));
3476     while ((cc2400_status() & FS_LOCK));
3477
3478     while (requested_mode == MODE_SPECAN) {
3479         for (f = low_freq; f < high_freq + 1; f++) {
3480             cc2400_set(FSDIV, f - 1);
3481             cc2400_strobe(SFSON);

```

```

3482     while (! ( cc2400_status () & FS_LOCK )) ;
3483     cc2400_strobe (SRX) ;
3484
3485     /* give the CC2400 time to acquire RSSI reading */
3486     volatile u32 j = 500; while (--j); /*FIXME crude delay */
3487     buf[3 * i] = (f >> 8) & 0xFF;
3488     buf[(3 * i) + 1] = f & 0xFF;
3489     buf[(3 * i) + 2] = cc2400_get (RSSI) >> 8;
3490     i++;
3491     if (i == 16) {
3492         enqueue (SPECAN, buf);
3493         i = 0;
3494
3495         handle_usb (clk_n);
3496     }
3497
3498     cc2400_strobe (SRFOFF);
3499     while ((cc2400_status () & FS_LOCK));
3500 }
3501 }
3502 RXLED_CLR;
3503 }
3504
3505 /*
3506 * Function: led_specan
3507 *
3508 * Run LED-based spectrum analyzer.
3509 */
3510 void led_specan (void) {
3511     int8_t lvl;
3512     u8 i = 0;
3513     u16 channels [3] = {2412, 2437, 2462};
3514     /*void (*set [3]) = {TXLED_SET, RXLED_SET, USRLED_SET}; */
3515     /*void (*clr [3]) = {TXLED_CLR, RXLED_CLR, USRLED_CLR}; */
3516 #ifdef UBERTOOTHTHONE
3517     PAEN_SET;
3518     /*HGM_SET; */
3519 #endif
3520     cc2400_set (LMTST, 0x2b22);
3521     cc2400_set (MDMTST0, 0x134b); /* without PRNG */
3522     cc2400_set (GRMDM, 0x0101); /* un-buffered mode, GFSK */
3523     cc2400_set (MDMCTRL, 0x0029); /* 160 kHz frequency deviation */
3524     cc2400_set (RSSI, 0x00F1); /* RSSI Sample over 2 symbols */
3525
3526     while (! (cc2400_status () & XOSC16M_STABLE));
3527     while ((cc2400_status () & FS_LOCK));
3528
3529     while (requested_mode == MODE_LED_SPECAN) {
3530         cc2400_set (FSDIV, channels [i] - 1);
3531         cc2400_strobe (SFSON);
3532         while (! (cc2400_status () & FS_LOCK));
3533         cc2400_strobe (SRX);
3534
3535         /* give the CC2400 time to acquire RSSI reading */
3536         volatile u32 j = 500; while (--j); /*FIXME crude delay */
3537         lvl = (int8_t)((cc2400_get (RSSI) >> 8) & 0xff);
3538         if (lvl > rssi_threshold) {

```

```

3539     switch ( i ) {
3540         case 0:
3541             TXLED_SET;
3542             break;
3543         case 1:
3544             RXLED_SET;
3545             break;
3546         case 2:
3547             USRLED_SET;
3548             break;
3549     }
3550 }
3551 else {
3552     switch ( i ) {
3553         case 0:
3554             TXLED_CLR;
3555             break;
3556         case 1:
3557             RXLED_CLR;
3558             break;
3559         case 2:
3560             USRLED_CLR;
3561             break;
3562     }
3563 }
3564
3565 i = ( i+1 ) % 3;
3566
3567 handle_usb( clk_n );
3568
3569 cc2400_strobe(SRFOFF);
3570 while ( ( cc2400_status() & FS_LOCK ) );
3571 }
3572 }
3573
3574 /*
3575 * Function: le_multi_reset_interval_timer
3576 *
3577 * Reset the connection's interval timer based on whether an anchor
3578 * and/or epoch are present.
3579 * link: target link
3580 */
3581 void le_multi_reset_interval_timer( le_state_t *l ) {
3582     u32 now = CLK100NS;
3583
3584     /* Calculate window_widening for the current link , */
3585     /* accounting for a clock reset */
3586     if ( l->next_expected_event < l->last_confirmed_event ){
3587         l->window_widening =
3588             CLK100NS_MAX - l->last_confirmed_event + l->next_expected_event;
3589     } else l->window_widening =
3590         l->next_expected_event - l->last_confirmed_event;
3591     l->window_widening =
3592         l->window_widening * (UBERTOOTH_CLOCK_PPM + l->sca);
3593     l->window_widening = DIVIDE_CEIL(l->window_widening, 1000000);
3594
3595     u32 next_minus_window = 0;
3596     /* IT = (nextEventStart - wideningWindow) - now; */

```

```

3597 /* Handle clock rollover cases , first for the window, then for the
3598 * current time "now" */
3599 if ( l->next_expected_event < l->window_widening ) {
3600     next_minus_window =
3601     CLK100NS_MAX - l->window_widening + l->next_expected_event;
3602 } else next_minus_window =
3603     l->next_expected_event - l->window_widening;
3604
3605 if ( next_minus_window < now ) {
3606     l->interval_timer = (CLK100NS_MAX - now) + next_minus_window;
3607 } else l->interval_timer = (next_minus_window - now);
3608
3609 /* Adjust for 312.5us interrupts , truncating down... we want to be
3610 * on the short side */
3611 l->interval_timer = l->interval_timer/3125;
3612
3613 /* Account for the fact that we observe the packet at its END. */
3614 l->interval_timer = l->interval_timer - PACKET_OFFSET;
3615 }
3616
3617 /*
3618 * Function: le_multi_cleanup
3619 * _____
3620 * Gets rid of stale connections
3621 */
3622 void le_multi_cleanup(void){
3623     /* Drop any stale connections */
3624     le_state_t *l = NULL;
3625     u32 now = CLK100NS, elapsed = 0;
3626     for (int i = 0; i < MAX_LINKS; i++){
3627         l = &(le_hdlr.links[i]);
3628         if (l->link_state != LINK_INACTIVE) {
3629             /* Our window is just too wide... get rid of the connection! */
3630             if (l->window_widening >= l->conn_interval/4){
3631                 reset_le(l);
3632                 do_hop = 1; /* in case this connection was next/active */
3633             }
3634
3635             /* Calculate time elapsed since the last packet */
3636             if ( now < l->last_packet )
3637                 elapsed = (CLK100NS_MAX - l->last_packet) + now;
3638             else elapsed = now - l->last_packet;
3639
3640             /* It's been too long since the last packet,
3641             * get rid of the connection */
3642             if (elapsed >= MAX_INACTIVE_LINK_TIME*10000000){
3643                 reset_le(l);
3644                 do_hop = 1; /* in case this connection was next/active */
3645             }
3646         }
3647     }
3648 }
3649
3650 /*
3651 * Function: le_multi_update_adv_state
3652 * _____
3653 * Progress the advertisement state when the timer expires .

```

```

3654 */
3655 void le_multi_update_adv_state(void) {
3656     le_state_t *a = &(le_hdlr.adv_link);
3657
3658     if (le_hdlr.cur_adv_state == ADV_CANDIDATE) {
3659         if (a->channel_idx == 37) {
3660             a->channel_idx = 38;
3661             le_hdlr.adv_timer = 32; /* 10 ms */
3662         } else if (a->channel_idx == 38) {
3663             a->channel_idx = 39;
3664             le_hdlr.adv_timer = 32; /* 10 ms */
3665         } else {
3666             a->channel_idx = 37;
3667             le_hdlr.cur_adv_state = ADV_SEARCH;
3668             le_hdlr.adv_timer_en = 0;
3669         }
3670     } else {
3671         a->channel_idx = 37;
3672         le_hdlr.adv_timer_en = 0;
3673     }
3674 }
```

1.4 firmware/bluetooth_rxtx/ubertooth_clock.h

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2015 Hannes Ellinger
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #ifndef _UBERTOOTHCLOCK_H
22 #define _UBERTOOTHCLOCK_H value
23
24 #include "inttypes.h"
25
26 /*
27  * CLK100NS is a free-running clock with a period of 100 ns. It resets
28  * every  $2^{15} \times 10^5$  cycles (about 5.5 minutes) – computed from clkn
29  * and timer0 (T0TC)
30  *
31  * clkn is the native (local) clock as defined in the Bluetooth
32  * specification. It advances 3200 times per second. Two clkn periods
33  * make a Bluetooth time slot.
34 */
35 #define UBERTOOTHCLOCK_PPM 20
36 #define CLK100NS_MAX 3276800000
37
38 volatile uint32_t clkn;
39 volatile uint32_t last_hop;
40
41 volatile uint32_t clkn_offset;
42 volatile uint16_t clk100ns_offset;
43
44 /* linear clock drift */
45 volatile int16_t clk_drift_ppm;
46 volatile uint16_t clk_drift_correction;
47
48 volatile uint32_t clkn_last_drift_fix;
49 volatile uint32_t clkn_next_drift_fix;
50
51 #define CLK100NS (3125*(clkn & 0xffff) + T0TC)
52 #define LE_BASECLK (12500) /* 1.25 ms in units of 100ns */
53
54 void clkn_stop();
55 void clkn_start();
```

```
56 void clk_n_init();  
57  
58 #endif
```

1.5 firmware/common/ubertooth.h

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2010, 2011 Michael Ossmann
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #ifndef _UBERTOOTH_H
22 #define _UBERTOOTH_H
23
24 #include "lpc17.h"
25 #include "types.h"
26 #include "cc2400.h"
27 #include "ubertooth_interface.h"
28
29 typedef void (*IAP_ENTRY)(u32 [], u32 []);
30 extern const IAP_ENTRY iap_entry;
31
32 /* operating modes */
33 enum operating_modes {
34     MODE_IDLE = 0,
35     MODE_RX_SYMBOLS = 1,
36     MODE_TX_SYMBOLS = 2,
37     MODE_TX_TEST = 3,
38     MODE_SPECAN = 4,
39     MODE_RANGE_TEST = 5,
40     MODE_REPEATERS = 6,
41     MODE_LED_SPECAN = 7,
42     MODE_BT_FOLLOW = 8,
43     MODE_BT_FOLLOW_LE = 9,
44     MODE_BT_MULTIFOLLOW_LE = 10, /* */
45     MODE_BT_PROMISC_LE = 11,
46     MODE_RESET = 12,
47     MODE_BT_SLAVE_LE = 13,
48     MODE_EGO = 14,
49     MODE_AFH = 15,
50     MODE_RX_GENERIC = 16,
51     MODE_TX_GENERIC = 17,
52 };
53
54 /* hardware identification number */
55 #define BOARD_ID_UBERTOOTH_ZERO 0
```

```

56 #define BOARD_ID_UBERTOOTH_ONE 1
57 #define BOARD_ID_TC13BADGE 2
58
59 #ifdef UBERTOOTH_ZERO
60 #define BOARD_ID BOARD_ID_UBERTOOTH_ZERO
61 #endif
62 #ifdef UBERTOOTH_ONE
63 #define BOARD_ID BOARD_ID_UBERTOOTH_ONE
64 #endif
65 #ifdef TC13BADGE
66 #define BOARD_ID BOARD_ID_TC13BADGE
67 #endif
68
69 /* GPIO pins */
70 #ifdef UBERTOOTH_ZERO
71 #define PINUSRLED (1 << 11) /* P0.11 */
72 #define PIN_RXLED (1 << 28) /* P4.28 */
73 #define PIN_TXLED (1 << 29) /* P4.29 */
74 #define PIN_CC1V8 (1 << 29) /* P1.29 */
75 #define PIN_CC3V3 (1 << 0 ) /* P1.0 */
76 #define PIN_VBUS (1 << 30) /* P1.30 */
77 #define PIN_RX (1 << 1 ) /* P1.1 */
78 #define PIN_TX (1 << 4 ) /* P1.4 */
79 #define PIN_CSN (1 << 8 ) /* P1.8 */
80 #define PIN_SCLK (1 << 9 ) /* P1.9 */
81 #define PIN_MOSI (1 << 10) /* P1.10 */
82 #define PIN_MISO (1 << 14) /* P1.14 */
83 #define PIN_GIO6 (1 << 15) /* P1.15 */
84 #define PIN_BTGR (1 << 31) /* P1.31 */
85 #define PIN_SSEL0 (1 << 9 ) /* P2.9 */
86 #endif
87 #ifdef UBERTOOTH_ONE
88 #define PINUSRLED (1 << 1 ) /* P1.1 */
89 #define PIN_RXLED (1 << 4 ) /* P1.4 */
90 #define PIN_TXLED (1 << 8 ) /* P1.8 */
91 #define PIN_CC1V8 (1 << 9 ) /* P1.9 */
92 #define PIN_CC3V3 (1 << 14) /* P1.14 */
93 #define PIN_VBUS (1 << 30) /* P1.30 */
94 #define PIN_RX (1 << 15) /* P1.15 */
95 #define PIN_TX (1 << 29) /* P4.29 */
96 #define PIN_CSN (1 << 5 ) /* P2.5 */
97 #define PIN_SCLK (1 << 4 ) /* P2.4 */
98 #define PIN_MOSI (1 << 0 ) /* P2.0 */
99 #define PIN_MISO (1 << 1 ) /* P2.1 */
100 #define PIN_GIO6 (1 << 2 ) /* P2.2 */
101 #define PIN_BTGR (1 << 10) /* P1.10 */
102 #define PIN_SSEL1 (1 << 28) /* P4.28 */
103 #define PIN_PAEN (1 << 7 ) /* P2.7 */
104 #define PIN_HGM (1 << 8 ) /* P2.8 */
105 #endif
106 #ifdef TC13BADGE
107 #define PIN_CC1V8 (1 << 0 ) /* P1.0 */
108 #define PIN_CC3V3 (1 << 14) /* P1.14 */
109 #define PIN_VBUS (1 << 30) /* P1.30 */
110 #define PIN_DIGITAL2 (1 << 0 ) /* P2.0 */
111 #define PIN_DIGITAL3 (1 << 1 ) /* P2.1 */

```

```

112 #define PIN_DIGITAL4 (1 << 2 ) /* P2.2 */
113 #define PIN_DIGITAL5 (1 << 3 ) /* P2.3 */
114 #define PIN_DIGITAL6 (1 << 4 ) /* P2.4 */
115 #define PIN_DIGITAL7 (1 << 5 ) /* P2.5 */
116 #define PIN_DIGITAL8 (1 << 6 ) /* P2.6 */
117 #define PIN_DIGITAL9 (1 << 7 ) /* P2.7 */
118 #define PIN_SW1 (1 << 8 ) /* P2.8 */
119 #define PIN_CSN (1 << 1 ) /* P1.1 */
120 #define PIN_SCLK (1 << 4 ) /* P1.4 */
121 #define PIN_MOSI (1 << 8 ) /* P1.8 */
122 #define PIN_MISO (1 << 9 ) /* P1.9 */
123 #define PIN_GIO6 (1 << 10) /* P1.10 */
124 #define PIN_SSEL1 (1 << 28) /* P4.28 */
125 #define PIN_R8C_CTL (1 << 22) /* P1.22 connects to R8C's P4.5 */
126 #define PIN_R8C_ACK (1 << 19) /* P1.19 connects to R8C's P0.0 */
127 /* RX, TX, and BT/GR are fixed to ground on TC13BADGE */
128 #endif
129
130 /* indicator LED control */
131 #ifdef UBERTOOTHTZERO
132 #define USRLED (FIO0PIN & PINUSRLED)
133 #define USRLED_SET (FIO0SET = PINUSRLED)
134 #define USRLED_CLR (FIO0CLR = PINUSRLED)
135 #define RXLED (FIO4PIN & PIN_RXLED)
136 #define RXLED_SET (FIO4SET = PIN_RXLED)
137 #define RXLED_CLR (FIO4CLR = PIN_RXLED)
138 #define TXLED (FIO4PIN & PIN_TXLED)
139 #define TXLED_SET (FIO4SET = PIN_TXLED)
140 #define TXLED_CLR (FIO4CLR = PIN_TXLED)
141 #endif
142 #ifdef UBERTOOTHLONE
143 #define USRLED (FIO1PIN & PINUSRLED)
144 #define USRLED_SET (FIO1SET = PINUSRLED)
145 #define USRLED_CLR (FIO1CLR = PINUSRLED)
146 #define RXLED (FIO1PIN & PIN_RXLED)
147 #define RXLED_SET (FIO1SET = PIN_RXLED)
148 #define RXLED_CLR (FIO1CLR = PIN_RXLED)
149 #define TXLED (FIO1PIN & PIN_TXLED)
150 #define TXLED_SET (FIO1SET = PIN_TXLED)
151 #define TXLED_CLR (FIO1CLR = PIN_TXLED)
152 #endif
153 #ifdef TC13BADGE
154 /*FIXME The LEDs need to be controlled by talking to the R8C. */
155 #define USRLED 0
156 #define USRLED_SET
157 #define USRLED_CLR
158 #define RXLED 0
159 #define RXLED_SET
160 #define RXLED_CLR
161 #define TXLED 0
162 #define TXLED_SET
163 #define TXLED_CLR
164 #endif
165
166 /* SW1 button press */
167 #ifdef TC13BADGE

```

```

168 #define SW1 (FIO2PIN & PIN_SW1)
169 #endif
170
171 /* R8C control */
172 #ifdef TC13BADGE
173 #define R8C_CTL_SET (FIO1SET = PIN_R8C_CTL)
174 #define R8C_CTL_CLR (FIO1CLR = PIN_R8C_CTL)
175 #define R8C_ACK (FIO1PIN & PIN_R8C_ACK)
176 #endif
177
178 /* SSEL (SPI slave select) control for CC2400 DIO
179 * (un-buffered) serial */
180 #ifdef UBERTOOTHZERO
181 #define DIO_SSEL_SET (FIO2SET = PIN_SSEL0)
182 #define DIO_SSEL_CLR (FIO2CLR = PIN_SSEL0)
183 #endif
184 #if defined UBERTOOTHZONE || defined TC13BADGE
185 #define DIO_SSEL_SET (FIO4SET = PIN_SSEL1)
186 #define DIO_SSEL_CLR (FIO4CLR = PIN_SSEL1)
187 #endif
188
189 /* 1V8 regulator control */
190 #define CC1V8 (FIO1PIN & PIN_CC1V8)
191 #define CC1V8_SET (FIO1SET = PIN_CC1V8)
192 #define CC1V8_CLR (FIO1CLR = PIN_CC1V8)
193
194 /* CC2400 control */
195 #ifdef UBERTOOTHZERO
196 #define CC3V3_SET (FIO1SET = PIN_CC3V3)
197 #define CC3V3_CLR (FIO1CLR = PIN_CC3V3)
198 #define RX_SET (FIO1SET = PIN_RX)
199 #define RX_CLR (FIO1CLR = PIN_RX)
200 #define TX_SET (FIO1SET = PIN_TX)
201 #define TX_CLR (FIO1CLR = PIN_TX)
202 #define CSN_SET (FIO1SET = PIN_CSN)
203 #define CSN_CLR (FIO1CLR = PIN_CSN)
204 #define SCLK_SET (FIO1SET = PIN_SCLK)
205 #define SCLK_CLR (FIO1CLR = PIN_SCLK)
206 #define MOSI_SET (FIO1SET = PIN_MOSI)
207 #define MOSI_CLR (FIO1CLR = PIN_MOSI)
208 #define GIO6 (FIO2PIN & PIN_GIO6)
209 #define GIO6_SET (FIO1SET = PIN_GIO6)
210 #define GIO6_CLR (FIO1CLR = PIN_GIO6)
211 #define BTGR_SET (FIO1SET = PIN_BTGR)
212 #define BTGR_CLR (FIO1CLR = PIN_BTGR)
213 #define MISO (FIO1PIN & PIN_MISO)
214 #endif
215 #ifdef UBERTOOTHZONE
216 #define CC3V3_SET (FIO1SET = PIN_CC3V3)
217 #define CC3V3_CLR (FIO1CLR = PIN_CC3V3)
218 #define RX_SET (FIO1SET = PIN_RX)
219 #define RX_CLR (FIO1CLR = PIN_RX)
220 #define TX_SET (FIO4SET = PIN_TX)
221 #define TX_CLR (FIO4CLR = PIN_TX)
222 #define CSN_SET (FIO2SET = PIN_CSN)
223 #define CSN_CLR (FIO2CLR = PIN_CSN)

```

```

224 #define SCLK_SET      (FIO2SET = PIN_SCLK)
225 #define SCLK_CLR      (FIO2CLR = PIN_SCLK)
226 #define MOSL_SET      (FIO2SET = PIN_MOSI)
227 #define MOSLCLR       (FIO2CLR = PIN_MOSI)
228 #define GIO6           (FIO2PIN & PIN_GIO6)
229 #define GIO6_SET       (FIO2SET = PIN_GIO6)
230 #define GIO6_CLR        (FIO2CLR = PIN_GIO6)
231 #define BTGR_SET       (FIO1SET = PIN_BTGR)
232 #define BTGR_CLR        (FIO1CLR = PIN_BTGR)
233 #define MISO            (FIO2PIN & PIN_MISO )
234 #endif
235 #ifdef TC13BADGE
236 #define CC3V3_SET      (FIO1SET = PIN_CC3V3)
237 #define CC3V3_CLR      (FIO1CLR = PIN_CC3V3)
238 #define CSN_SET         (FIO1SET = PIN_CSN)
239 #define CSN_CLR          (FIO1CLR = PIN_CSN)
240 #define SCLK_SET        (FIO1SET = PIN_SCLK)
241 #define SCLK_CLR         (FIO1CLR = PIN_SCLK)
242 #define MOSLSET         (FIO1SET = PIN_MOSI)
243 #define MOSLCLR         (FIO1CLR = PIN_MOSI)
244 #define GIO6             (FIO2PIN & PIN_GIO6)
245 #define GIO6_SET         (FIO1SET = PIN_GIO6)
246 #define GIO6_CLR          (FIO1CLR = PIN_GIO6)
247 #define MISO             (FIO1PIN & PIN_MISO )
248 #endif
249 /*
250  * DIO_SSP is the SSP assigned to the CC2400's secondary
251  * ("un-buffered") serial interface
252  */
253 #ifdef UBERTOOTHTHZERO
254 #define DIO_SSP_CR0     SSP0CR0
255 #define DIO_SSP_CR1     SSP0CR1
256 #define DIO_SSP_DR      SSP0DR
257 #define DIO_SSP_DMACR   SSP0DMACR
258 #define DIO_SSP_SRC     (1 << 1) /* for DMACCxConfig register */
259 #endif
260 #if defined UBERTOOTHTHONE || defined TC13BADGE
261 #define DIO_SSP_CR0     SSP1CR0
262 #define DIO_SSP_CR1     SSP1CR1
263 #define DIO_SSP_DR      SSP1DR
264 #define DIO_SSP_DMACR   SSP1DMACR
265 #define DIO_SSP_SRC     (3 << 1) /* for DMACCxConfig register */
266 #endif
267 #endif
268 /*
269  * CC2591 control */
270 #ifdef UBERTOOTHTHONE
271 #define PAEN            (FIO2PIN & PIN_PAEN)
272 #define PAEN_SET         (FIO2SET = PIN_PAEN)
273 #define PAEN_CLR         (FIO2CLR = PIN_PAEN)
274 #define HGM              (FIO2PIN & PIN_HGM)
275 #define HGML_SET         (FIO2SET = PIN_HGM)
276 #define HGMLCLR         (FIO2CLR = PIN_HGM)
277 #endif
278 /*
279  * USB VBUS monitoring */

```

```

280 #define VBUS (FIO1PIN & PIN_VBUS)
281
282 /*
283 * clock configuration
284 *
285 * main oscillator: 16 MHz (from CC2400)
286 * CPU clock (PLL0): 100 MHz
287 * USB clock (PLL1): 48 MHz
288 *
289 * The ToorCon 13 badge is configured with a 30 MHz CPU clock instead
290 * of 100 MHz to reduce heat at the voltage regulator. This is a
291 * sufficient clock speed for passive Bluetooth monitoring.
292 */
293 #ifdef TC13BADGE
294 #define MSEL0 14
295 #define NSEL0 0
296 #define CCLKSEL 15
297 #else
298 #define MSEL0 24
299 #define NSEL0 1
300 #define CCLKSEL 3
301 #endif
302 #define MSEL1 34
303 #define PSEL1 0
304
305 /* flash accelerator configuration */
306 #define FLASHTIM 0x4 /* up to 100 MHz CPU clock */
307
308 /*
309 * bootloader_ctrl is a fixed memory location used for passing
310 * information from the application to the bootloader across a reset
311 */
312 extern uint32_t bootloader_ctrl;
313 #define DFUMODE 0x4305BB21
314
315 void wait(u8 seconds);
316 void wait_ms(u32 ms);
317 void wait_us(u32 us);
318 u32 rbit(u32 value);
319 void gpio_init(void);
320 void all_pins_off(void);
321 void ubertooth_init(void);
322 void dio_ssp_init(void);
323 void atest_init(void);
324 void cc2400_init(void);
325 u32 cc2400_spi(u8 len, u32 data);
326 u16 cc2400_get(u8 reg);
327 void cc2400_set(u8 reg, u16 val);
328 u8 cc2400_get8(u8 reg);
329 void cc2400_set8(u8 reg, u8 val);
330 void cc2400_fifo_write(u8 len, u8 *data);
331 void cc2400_fifo_read(u8 len, u8 *buf);
332 u8 cc2400_status(void);
333 u8 cc2400_strobe(u8 reg);
334 void cc2400_reset(void);
335 void clock_start(void);
336 void reset(void);

```

```
337 void r8c_takeover(void);
338 void cc2400_tune_rx(uint16_t channel);
339 void cc2400_tune_tx(uint16_t channel);
340 void cc2400_hop_rx(uint16_t channel);
341 void cc2400_hop_tx(uint16_t channel);
342 void get_part_num(uint8_t *buffer, int *len);
343 void get_device_serial(uint8_t *buffer, int *len);
344 void set_isp(void);
345
346 #endif /* _UBERTOOTH_H */
```

1.6 host/libubertooth/src/ubertooth_control.c

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2010–2013 Michael Ossmann, Dominic Spill
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110–1301, USA.
19  */
20
21 #include <string.h>
22 #include <btbb.h>
23 #include "ubertooth_control.h"
24
25 void show_libusb_error(int error_code)
26 {
27     char *error_hint = "";
28     const char *error_name;
29
30     /* Available only in libusb > 1.0.3 */
31     /* error_name = libusb_error_name(error_code); */
32
33     switch (error_code) {
34     case LIBUSB_ERROR_TIMEOUT:
35         error_name="Timeout";
36         break;
37     case LIBUSB_ERROR_NO_DEVICE:
38         error_name="No Device";
39         error_hint="Check Ubertooth is connected to host";
40         break;
41     case LIBUSB_ERROR_ACCESS:
42         error_name="Insufficient Permissions";
43         break;
44     case LIBUSB_ERROR_OVERFLOW:
45         error_name="Overflow";
46         error_hint="Try resetting the Ubertooth";
47         break;
48     default:
49         error_name="Command Error";
50         break;
51     }
52
53     fprintf(stderr,"libUSB Error: %s: %s (%d)\n",
54             error_name, error_hint, error_code);
55 }
```

```

56
57 static void callback(struct libusb_transfer* transfer)
58 {
59     if(transfer->status != 0) {
60         show_libusb_error(transfer->status);
61     }
62     libusb_free_transfer(transfer);
63 }
64
65 void cmd_trim_clock(struct libusb_device_handle* devh, uint16_t offset)
66 {
67     uint8_t data[2] = {
68         (offset >> 8) & 0xff,
69         (offset >> 0) & 0xff
70     };
71
72     ubertooth_cmd_async(devh, CTRL_OUT, UBERTOOTH_TRIM_CLOCK, data, 2);
73 }
74
75 void cmd_fix_clock_drift(struct libusb_device_handle* devh, int16_t ppm)
76 {
77     uint8_t data[2] = {
78         (ppm >> 8) & 0xff,
79         (ppm >> 0) & 0xff
80     };
81
82     ubertooth_cmd_async(devh, CTRL_OUT, UBERTOOTH_FIX_CLOCK_DRIFT, data, 2);
83 }
84
85 int cmd_ping(struct libusb_device_handle* devh)
86 {
87     int r;
88
89     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_PING, 0, 0,
90                                 NULL, 0, 1000);
91     if (r < 0) {
92         show_libusb_error(r);
93         return r;
94     }
95     return 0;
96 }
97
98 int cmd_rx_syms(struct libusb_device_handle* devh)
99 {
100    int r;
101
102    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_RX_SYMBOLS, 0, 0,
103                                NULL, 0, 1000);
104    if (r < 0) {
105        show_libusb_error(r);
106        return r;
107    }
108    return 0;
109 }
110
111 int cmd_tx_syms(struct libusb_device_handle* devh)
112 {

```

```

113     return \
114         ubertooth_cmd_sync(devh, CTRL_OUT, UBERTOOTH_TX_SYMBOLS, 0, 0);
115 }
116
117 int cmd_specan(struct libusb_device_handle* devh, \
118                 u16 low_freq, u16 high_freq)
119 {
120     int r;
121
122     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SPECAN,
123                                 low_freq, high_freq, NULL, 0, 1000);
124     if (r < 0) {
125         show_libusb_error(r);
126         return r;
127     }
128     return 0;
129 }
130
131 int cmd_led_specan(struct libusb_device_handle* devh, \
132                     u16 rss_i_threshold)
133 {
134     int r;
135
136     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_LED_SPECAN,
137                                 rss_i_threshold, 0, NULL, 0, 1000);
138     if (r < 0) {
139         show_libusb_error(r);
140         return r;
141     }
142     return 0;
143 }
144
145 int cmd_set_usrled(struct libusb_device_handle* devh, u16 state)
146 {
147     int r;
148
149     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SETUSRLED, \
150                                 state, 0, NULL, 0, 1000);
151     if (r < 0) {
152         show_libusb_error(r);
153         return r;
154     }
155     return 0;
156 }
157
158 int cmd_get_usrled(struct libusb_device_handle* devh)
159 {
160     u8 state;
161     int r;
162
163     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GETUSRLED, 0, \
164                                 0, &state, 1, 1000);
165     if (r < 0) {
166         show_libusb_error(r);
167         return r;
168     }
169     return state;
170 }

```

```

171
172 int cmd_set_rxled(struct libusb_device_handle* devh, u16 state)
173 {
174     int r;
175
176     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_RXLED, \
177                                 state, 0, NULL, 0, 1000);
178     if (r < 0) {
179         show_libusb_error(r);
180         return r;
181     }
182     return 0;
183 }
184
185 int cmd_get_rxled(struct libusb_device_handle* devh)
186 {
187     u8 state;
188     int r;
189
190     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_RXLED, 0, 0,
191                                &state, 1, 1000);
192     if (r < 0) {
193         show_libusb_error(r);
194         return r;
195     }
196     return state;
197 }
198
199 int cmd_set_txled(struct libusb_device_handle* devh, u16 state)
200 {
201     int r;
202
203     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_TXLED, \
204                                 state, 0, NULL, 0, 1000);
205     if (r < 0) {
206         show_libusb_error(r);
207         return r;
208     }
209     return 0;
210 }
211
212 int cmd_get_txled(struct libusb_device_handle* devh)
213 {
214     u8 state;
215     int r;
216
217     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_TXLED, 0, \
218                                0, &state, 1, 1000);
219     if (r < 0) {
220         show_libusb_error(r);
221         return r;
222     }
223     return state;
224 }
225
226 int cmd_get_modulation(struct libusb_device_handle* devh)
227 {
228     u8 modulation;
229     int r;

```

```

230
231     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHLGETMOD, 0, 0,
232         &modulation, 1, 1000);
233     if (r < 0) {
234         show_libusb_error(r);
235         return r;
236     }
237
238     return modulation;
239 }
240
241 int cmd_get_channel(struct libusb_device_handle* devh)
242 {
243     u8 result[2];
244     int r;
245     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHLGETCHANNEL, \
246         0, 0, result, 2, 1000);
247     if (r == LIBUSB_ERROR_PIPE) {
248         fprintf(stderr, "control message unsupported\n");
249         return r;
250     } else if (r < 0) {
251         show_libusb_error(r);
252         return r;
253     }
254
255     return result[0] | (result[1] << 8);
256 }
257
258
259 int cmd_set_channel(struct libusb_device_handle* devh, u16 channel)
260 {
261     int r;
262
263     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHSETCHANNEL, \
264         channel, 0, NULL, 0, 1000);
265     if (r == LIBUSB_ERROR_PIPE) {
266         fprintf(stderr, "control message unsupported\n");
267         return r;
268     } else if (r < 0) {
269         show_libusb_error(r);
270         return r;
271     }
272     return 0;
273 }
274
275 int cmd_get_partnum(struct libusb_device_handle* devh)
276 {
277     u8 result[5];
278     int r;
279
280     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHLGETPARTNUM, \
281         0, 0, result, 5, 1000);
282     if (r < 0) {
283         show_libusb_error(r);
284         return r;
285     }
286     if (result[0] != 0) {
287         fprintf(stderr, "result not zero: %d\n", result[0]);

```

```

288     return 0;
289 }
290     result[1] | (result[2] << 8) | \
291         (result[3] << 16) | (result[4] << 24);
292 }
293
294 void print_serial(u8 *serial , FILE *fileptr )
295 {
296     int i;
297     if(fileptr == NULL)
298         fileptr = stdout;
299
300     fprintf(fileptr , "Serial No: ");
301     for(i=1; i<17; i++)
302         fprintf(fileptr , "%02x" , serial[i]);
303     fprintf(fileptr , "\n");
304 }
305
306 int cmd_get_serial( struct libusb_device_handle* devh , u8 *serial )
307 {
308     int r;
309     r = libusb_control_transfer(devh , CTRL_IN , UBERTOOTH_GET_SERIAL , \
310         0 , 0 , serial , 17 , 1000);
311     if (r < 0) {
312         show_libusb_error(r);
313         return r;
314     }
315     if (serial[0] != 0) {
316         fprintf(stderr , "result not zero: %d\n" , serial[0]);
317         return 1;
318     }
319     return 0;
320 }
321
322 int cmd_set_modulation( struct libusb_device_handle* devh , u16 mod)
323 {
324     int r;
325
326     r = libusb_control_transfer(devh , CTRL_OUT , UBERTOOTH_SET_MOD , \
327         mod , 0 , NULL , 0 , 1000);
328     if (r == LIBUSB_ERROR_PIPE) {
329         fprintf(stderr , "control message unsupported\n");
330         return r;
331     } else if (r < 0) {
332         show_libusb_error(r);
333         return r;
334     }
335     return 0;
336 }
337
338 int cmd_set_isp( struct libusb_device_handle* devh )
339 {
340     int r;
341
342     r = libusb_control_transfer(devh , CTRL_OUT , UBERTOOTH_SET_ISP , 0 , 0 ,
343         NULL , 0 , 1000);
344     /* LIBUSB_ERROR_PIPE or LIBUSB_ERROR_OTHER is expected */
345     if (r && (r != LIBUSB_ERROR_PIPE) && (r != LIBUSB_ERROR_OTHER)) &&

```

```

346     (r != LIBUSB_ERROR_NO_DEVICE)) {
347         show_libusb_error(r);
348         return r;
349     }
350     return 0;
351 }
352
353 int cmd_reset(struct libusb_device_handle* devh)
354 {
355     int r;
356
357     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHLRESET, 0, 0,
358                                 NULL, 0, 1000);
359     /* LIBUSB_ERROR_PIPE or LIBUSB_ERROR_OTHER is expected */
360     if (r && (r != LIBUSB_ERROR_PIPE) && (r != LIBUSB_ERROR_OTHER) &&
361         (r != LIBUSB_ERROR_NO_DEVICE)) {
362         show_libusb_error(r);
363         return r;
364     }
365     return 0;
366 }
367
368 int cmd_stop(struct libusb_device_handle* devh)
369 {
370     int r;
371
372     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHLSTOP, 0, 0,
373                                 NULL, 0, 1000);
374     if (r == LIBUSB_ERROR_PIPE) {
375         fprintf(stderr, "control message unsupported\n");
376         return r;
377     } else if (r < 0) {
378         show_libusb_error(r);
379         return r;
380     }
381     return 0;
382 }
383
384 int cmd_set_paen(struct libusb_device_handle* devh, u16 state)
385 {
386     int r;
387
388     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_PAEN, \
389                                 state, 0, NULL, 0, 1000);
390     if (r == LIBUSB_ERROR_PIPE) {
391         fprintf(stderr, "control message unsupported\n");
392         return r;
393     } else if (r < 0) {
394         show_libusb_error(r);
395         return r;
396     }
397     return 0;
398 }
399
400 int cmd_set_hgm(struct libusb_device_handle* devh, u16 state)
401 {
402     int r;
403

```

```

404     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHSETHGM, \
405         state, 0, NULL, 0, 1000);
406     if (r == LIBUSB_ERROR_PIPE) {
407         fprintf(stderr, "control message unsupported\n");
408         return r;
409     } else if (r < 0) {
410         show_libusb_error(r);
411         return r;
412     }
413     return 0;
414 }
415
416 int cmd_tx_test(struct libusb_device_handle* devh)
417 {
418     int r;
419
420     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHTXTEST, 0, 0,
421         NULL, 0, 1000);
422     if (r == LIBUSB_ERROR_PIPE) {
423         fprintf(stderr, "control message unsupported\n");
424         return r;
425     } else if (r < 0) {
426         show_libusb_error(r);
427         return r;
428     }
429     return 0;
430 }
431
432 int cmd_flash(struct libusb_device_handle* devh)
433 {
434     int r;
435
436     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHFLASH, 0, 0,
437         NULL, 0, 1000);
438     if (r != LIBUSB_SUCCESS) {
439         show_libusb_error(r);
440         return r;
441     }
442     return 0;
443 }
444
445 int cmd_get_palevel(struct libusb_device_handle* devh)
446 {
447     u8 level;
448     int r;
449
450     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHGETPALEVEL, \
451         0, 0, &level, 1, 3000);
452     if (r < 0) {
453         show_libusb_error(r);
454         return r;
455     }
456     return level;
457 }
458
459 int cmd_set_palevel(struct libusb_device_handle* devh, u16 level)
460 {
461     int r;

```

```

462
463     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHSERIALLEVEL, \
464         level, 0, NULL, 0, 3000);
465     if (r != LIBUSB_SUCCESS) {
466         if (r == LIBUSB_ERROR_PIPE) {
467             fprintf(stderr, "control message unsupported\n");
468         } else {
469             show_libusb_error(r);
470         }
471         return r;
472     }
473     return 0;
474 }
475
476 int cmd_get_rangeresult(struct libusb_device_handle* devh,
477                         rangetest_result *rr)
478 {
479     u8 result[5];
480     int r;
481
482     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHRANGE_CHECK, \
483         0, 0, result, sizeof(result), 3000);
484     if (r < LIBUSB_SUCCESS) {
485         if (r == LIBUSB_ERROR_PIPE) {
486             fprintf(stderr, "control message unsupported\n");
487         } else {
488             show_libusb_error(r);
489         }
490         return r;
491     }
492
493     rr->valid      = result[0];
494     rr->request_pa = result[1];
495     rr->request_num = result[2];
496     rr->reply_pa    = result[3];
497     rr->reply_num   = result[4];
498
499     return 0;
500 }
501
502 int cmd_range_test(struct libusb_device_handle* devh)
503 {
504     int r;
505
506     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHRANGE_TEST, \
507         0, 0, NULL, 0, 1000);
508     if (r != LIBUSB_SUCCESS) {
509         if (r == LIBUSB_ERROR_PIPE) {
510             fprintf(stderr, "control message unsupported\n");
511         } else {
512             show_libusb_error(r);
513         }
514         return r;
515     }
516     return 0;
517 }
518

```

```

519 int cmd_repeater( struct libusb_device_handle* devh)
520 {
521     int r;
522
523     r = libusb_control_transfer( devh , CTRLOUT, UBERTOOTHRPEATER, \
524         0, 0, NULL, 0, 1000);
525     if (r != LIBUSB_SUCCESS) {
526         if (r == LIBUSB_ERROR_PIPE) {
527             fprintf(stderr, "control message unsupported\n");
528         } else {
529             show_libusb_error(r);
530         }
531         return r;
532     }
533     return 0;
534 }
535
536 void cmd_get_rev_num( struct libusb_device_handle* devh , \
537     char *version , u8 len)
538 {
539     u8 result[2 + 1 + 255];
540     u16 result_ver;
541     int r;
542     r = libusb_control_transfer( devh , CTRL_IN, UBERTOOTHGET_REV_NUM, \
543         0, 0, result , sizeof(result) , 1000);
544     if (r == LIBUSB_ERROR_PIPE) {
545         fprintf(stderr, "control message unsupported\n");
546         snprintf(version , len - 1, "error: %d" , r);
547         version[len-1] = '\0';
548         return;
549     } else if (r < 0) {
550         show_libusb_error(r);
551         snprintf(version , len - 1, "error: %d" , r);
552         version[len-1] = '\0';
553         return;
554     }
555
556     result_ver = result[0] | (result[1] << 8);
557     if (r == 2) { /* old-style SVN rev */
558         sprintf(version , "%u" , result_ver);
559     } else {
560         len = MIN(r - 3, MIN(len - 1, result[2]));
561         memcpy(version , &result[3] , len);
562         version[len] = '\0';
563     }
564 }
565
566 void cmd_get_compile_info( struct libusb_device_handle* devh , \
567     char *compile_info , u8 len)
568 {
569     u8 result[1 + 255];
570     int r;
571     r = libusb_control_transfer( devh , CTRL_IN, \
572         UBERTOOTHGET_COMPILEINFO, 0, 0, result , sizeof(result) , 1000);
573     if (r == LIBUSB_ERROR_PIPE) {
574         fprintf(stderr, "control message unsupported\n");
575         snprintf(compile_info , len - 1, "error: %d" , r);

```

```

576     compile_info[len - 1] = '\0';
577     return;
578 } else if (r < 0) {
579     show_libusb_error(r);
580     snprintf(compile_info, len - 1, "error: %d", r);
581     compile_info[len - 1] = '\0';
582     return;
583 }
584
585 len = MIN(r - 1, MIN(len - 1, result[0]));
586 memcpy(compile_info, &result[1], len);
587 compile_info[len] = '\0';
588 }
589
590 int cmd_get_board_id(struct libusb_device_handle* devh)
591 {
592     u8 board_id;
593     int r;
594     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_BOARD_ID, \
595         0, 0, &board_id, 1, 1000);
596     if (r == LIBUSB_ERROR_PIPE) {
597         fprintf(stderr, "control message unsupported\n");
598         return r;
599     } else if (r < 0) {
600         show_libusb_error(r);
601         return r;
602     }
603
604     return board_id;
605 }
606
607 int cmd_set_squelch(struct libusb_device_handle* devh, u16 level)
608 {
609     int r;
610
611     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_SQUELCH, \
612         level, 0, NULL, 0, 3000);
613     if (r != LIBUSB_SUCCESS) {
614         if (r == LIBUSB_ERROR_PIPE) {
615             fprintf(stderr, "control message unsupported\n");
616         } else {
617             show_libusb_error(r);
618         }
619         return r;
620     }
621     return 0;
622 }
623
624 int cmd_get_squelch(struct libusb_device_handle* devh)
625 {
626     u8 level;
627     int r;
628
629     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_SQUELCH, \
630         0, 0, &level, 1, 3000);
631     if (r < 0) {
632         show_libusb_error(r);
633         return r;

```

```

634     }
635     return level;
636 }
637
638 int cmd_set_bdaddr(struct libusb_device_handle* devh, u64 address)
639 {
640     int r, data_len;
641     u64 syncword;
642     data_len = 16;
643     unsigned char data[data_len];
644
645     syncword = btbb_gen_syncword(address & 0xffffffff);
646     /*printf("syncword=%#llx\n", syncword); */
647     for(r=0; r < 8; r++)
648         data[r] = (address >> (8*r)) & 0xff;
649     for(r=0; r < 8; r++)
650         data[r+8] = (syncword >> (8*r)) & 0xff;
651
652     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHS_SET_BDADDR, \
653                                0, 0, data, data_len, 1000);
654     if (r < 0) {
655         if (r == LIBUSB_ERROR_PIPE) {
656             fprintf(stderr, "control message unsupported\n");
657         } else {
658             show_libusb_error(r);
659         }
660         return r;
661     } else if (r < data_len) {
662         fprintf(stderr, "Only %d of %d bytes transferred\n", r, data_len);
663         return 1;
664     }
665     return 0;
666 }
667
668 int cmd_start_hopping(struct libusb_device_handle* devh, \
669                       int clkns_offset, int clk100ns_offset)
670 {
671     int r;
672     uint8_t data[6];
673     for(r=0; r < 4; r++)
674         data[r] = (clkns_offset >> (8*(3-r))) & 0xff;
675
676     data[4] = (clk100ns_offset >> 8) & 0xff;
677     data[5] = (clk100ns_offset >> 0) & 0xff;
678
679     r = ubertooth_cmd_async(devh, CTRL_OUT, \
680                            UBERTOOTHS_START_HOPPING, data, 6);
681     if (r < 0) {
682         if (r == LIBUSB_ERROR_PIPE) {
683             fprintf(stderr, "control message unsupported\n");
684         } else {
685             show_libusb_error(r);
686         }
687         return r;
688     }
689     return 0;
690 }

```

```

691
692 int cmd_set_clock(struct libusb_device_handle* devh, u32 clkn)
693 {
694     int r;
695     u8 data[4];
696     for(r=0; r < 4; r++)
697         data[r] = (clkn >> (8*r)) & 0xff;
698
699     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHSSET_CLOCK, \
700                                 0, 0, data, 4, 1000);
701     if (r < 0) {
702         if (r == LIBUSB_ERROR_PIPE) {
703             fprintf(stderr, "control message unsupported\n");
704         } else {
705             show_libusb_error(r);
706         }
707         return r;
708     }
709     return 0;
710 }
711
712 uint32_t cmd_get_clock(struct libusb_device_handle* devh)
713 {
714     u32 clock = 0;
715     unsigned char data[4];
716     int r;
717
718     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTHSGET_CLOCK, 0, 0,
719                                data, 4, 3000);
720     if (r < 0) {
721         show_libusb_error(r);
722         return r;
723     }
724     clock = data[0] | data[1] << 8 | data[2] << 16 | data[3] << 24;
725     printf("Read clock = 0x%x\n", clock);
726     return clock;
727 }
728
729 int cmd_btlesniffing(struct libusb_device_handle* devh, u16 num)
730 {
731     int r;
732
733     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTHSBTLESNIFFING,\ \
734                                 num, 0, NULL, 0, 1000);
735     if (r < 0) {
736         if (r == LIBUSB_ERROR_PIPE) {
737             fprintf(stderr, "control message unsupported\n");
738         } else {
739             show_libusb_error(r);
740         }
741         return r;
742     }
743     return 0;
744 }
745
746 int cmd_btlemulti_sniffing(\
747     struct libusb_device_handle* devh, u16 num) {

```

```

748     printf("in the function to send multi\n");
749     int r;
750
751     r = libusb_control_transfer(devh, CTRL_OUT, \
752                                 UBERTOOTH_BTLE_MULTLSNIFFING, num, 0, NULL, 0, 1000);
753
754     printf("Returned from the control handler , r = %d\n", r);
755     if (r < 0) {
756         if (r == LIBUSB_ERROR_PIPE) {
757             fprintf(stderr, "control message unsupported\n");
758         } else {
759             show_libusb_error(r);
760         }
761         return r;
762     }
763     return 0;
764 }
765
766 int cmd_set_afh_map(struct libusb_device_handle* devh, \
767                      uint8_t* afh_map)
768 {
769     uint8_t buffer[LIBUSB_CONTROL_SETUP_SIZE+10];
770     struct libusb_transfer *xfer = libusb_alloc_transfer(0);
771
772     libusb_fill_control_setup(buffer, CTRL_OUT, \
773                               UBERTOOTH_SET_AFHMAP, \0, 0, 10);
774     memcpy (&buffer[LIBUSB_CONTROL_SETUP_SIZE], afh_map, 10 );
775     libusb_fill_control_transfer(xfer, devh, buffer, callback, \
776                                  NULL, 1000);
777     libusb_submit_transfer(xfer);
778
779     return 0;
780 }
781
782 int cmd_clear_afh_map(struct libusb_device_handle* devh)
783 {
784     int r;
785     r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_CLEAR_AFHMAP, \
786                                0, 0, NULL, 0, 1000);
787     if (r < 0) {
788         if (r == LIBUSB_ERROR_PIPE) {
789             fprintf(stderr, "control message unsupported\n");
790         } else {
791             show_libusb_error(r);
792         }
793         return r;
794     }
795     return 0;
796 }
797
798 u32 cmd_get_access_address(struct libusb_device_handle* devh)
799 {
800     u32 access_address = 0;
801     unsigned char data[4];
802     int r;
803
804     r = libusb_control_transfer(devh, CTRL_IN, \
805                                 UBERTOOTH_GET_ACCESS_ADDRESS, 0, 0, data, 4, 3000);

```

```

806     if (r < 0) {
807         show_libusb_error(r);
808         return r;
809     }
810     access_address = data[0] | data[1] << 8 | \
811                           data[2] << 16 | data[3] << 24;
812     return access_address;
813 }
814
815 int cmd_set_access_address(\n
816     struct libusb_device_handle* devh, u32 access_address)
817 {
818     int r;
819     u8 data[4];
820     for(r=0; r < 4; r++)
821         data[r] = (access_address >> (8*r)) & 0xff;
822
823     r = libusb_control_transfer(devh, CTRL_OUT, \
824                                 UBERTOOTHS_SET_ACCESS_ADDRESS, 0, 0, data, 4, 1000);
825     if (r < 0) {
826         if (r == LIBUSB_ERROR_PIPE) {
827             fprintf(stderr, "control message unsupported\n");
828         } else {
829             show_libusb_error(r);
830         }
831         return r;
832     }
833     return 0;
834 }
835
836 int cmd_do_something(struct libusb_device_handle *devh, \
837     unsigned char *data, int len)
838 {
839     int r = libusb_control_transfer(devh, CTRL_OUT, \
840                                     UBERTOOTHS_DO_SOMETHING, 0, 0, data, len, 1000);
841     if (r < 0) {
842         if (r == LIBUSB_ERROR_PIPE) {
843             fprintf(stderr, "control message unsupported\n");
844         } else {
845             show_libusb_error(r);
846         }
847         return r;
848     }
849     return 0;
850 }
851
852 int cmd_do_something_reply(\n
853     struct libusb_device_handle* devh, unsigned char *data, int len)
854 {
855     int r = libusb_control_transfer(devh, CTRL_IN, \
856                                     UBERTOOTHS_DO_SOMETHING_REPLY, 0, 0, data, len, 3000);
857     if (r < 0) {
858         if (r == LIBUSB_ERROR_PIPE) {
859             fprintf(stderr, "control message unsupported\n");
860         } else {
861             show_libusb_error(r);
862         }

```

```

863     return r;
864 }
865 return r;
866 }
867
868 int cmd_get_crc_verify(struct libusb_device_handle* devh)
869 {
870     u8 verify;
871     int r;
872
873     r = libusb_control_transfer(devh, CTRL_IN, \
874         UBERTOOTH_GET_CRC_VERIFY, 0, 0, &verify, 1, 1000);
875     if (r < 0) {
876         show_libusb_error(r);
877         return r;
878     }
879     return verify;
880 }
881
882 int cmd_set_crc_verify(struct libusb_device_handle* devh, int verify)
883 {
884     int r;
885
886     r = libusb_control_transfer(devh, CTRL_OUT, \
887         UBERTOOTH_SET_CRC_VERIFY, verify, 0, NULL, 0, 1000);
888     if (r < 0) {
889         show_libusb_error(r);
890         return r;
891     }
892     return 0;
893 }
894
895 int cmd_poll(struct libusb_device_handle* devh, usb_pkt_rx *p)
896 {
897     int r;
898
899     r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_POLL, 0, 0,
900         (u8 *)p, sizeof(usb_pkt_rx), 1000);
901     if (r < 0) {
902         show_libusb_error(r);
903         return r;
904     }
905     return r;
906 }
907
908 int cmd_btle_promisc(struct libusb_device_handle* devh)
909 {
910     int r;
911
912     r = libusb_control_transfer(devh, CTRL_OUT, \
913         UBERTOOTH_BTLE_PROMISC, 0, 0, NULL, 0, 1000);
914     if (r < 0) {
915         if (r == LIBUSB_ERROR_PIPE) {
916             fprintf(stderr, "control message unsupported\n");
917         } else {
918             show_libusb_error(r);
919         }
920     }
921     return r;

```

```

921     }
922     return 0;
923 }
924
925 int cmd_read_register(struct libusb_device_handle* devh, u8 reg)
926 {
927     int r;
928     u8 data[2];
929
930     r = libusb_control_transfer(devh, CTRL_IN, \
931         UBERTOOTHSREADREGISTER, reg, 0, data, 2, 1000);
932     if (r < 0) {
933         if (r == LIBUSB_ERROR_PIPE) {
934             fprintf(stderr, "control message unsupported\n");
935         } else {
936             show_libusb_error(r);
937         }
938         return r;
939     }
940
941     return (data[0] << 8) | data[1];
942 }
943
944 int cmd_btleslave(struct libusb_device_handle* devh, u8 *mac_address)
945 {
946     int r;
947
948     r = libusb_control_transfer(devh, CTRLOUT, \
949         UBERTOOTHSBTLESLAVE, 0, 0, mac_address, 6, 1000);
950     if (r < 0) {
951         if (r == LIBUSB_ERROR_PIPE) {
952             fprintf(stderr, "control message unsupported\n");
953         } else {
954             show_libusb_error(r);
955         }
956         return r;
957     }
958
959     return 0;
960 }
961
962 int cmd_btlesettarget(\n
963     struct libusb_device_handle* devh, u8 *mac_address)
964 {
965     int r;
966
967     r = libusb_control_transfer(devh, CTRLOUT, \
968         UBERTOOTHSBTLESETTARGET, 0, 0, mac_address, 6, 1000);
969     if (r < 0) {
970         if (r == LIBUSB_ERROR_PIPE) {
971             fprintf(stderr, "control message unsupported\n");
972         } else {
973             show_libusb_error(r);
974         }
975         return r;
976     }
977     return 0;
978 }

```

```

979
980 int cmd_set_jam_mode( struct libusb_device_handle* devh , int mode) {
981     int r ;
982
983     r = libusb_control_transfer( devh , CTRLOUT, \
984                                 UBERTOOTHLJAMMODE, mode, 0, NULL, 0, 1000);
985     if (r < 0) {
986         if (r == LIBUSB_ERROR_PIPE) {
987             fprintf(stderr , "control message unsupported\n");
988         } else {
989             show_libusb_error(r);
990         }
991         return r;
992     }
993     return 0;
994 }
995
996 int cmd_ego( struct libusb_device_handle* devh , int mode)
997 {
998     int r ;
999
1000    r = libusb_control_transfer( devh , CTRLOUT, UBERTOOTHEGO, mode, 0,
1001                                NULL, 0, 1000);
1002    if (r < 0) {
1003        if (r == LIBUSB_ERROR_PIPE) {
1004            fprintf(stderr , "control message unsupported\n");
1005        } else {
1006            show_libusb_error(r);
1007        }
1008        return r;
1009    }
1010    return 0;
1011 }
1012
1013 int cmd_afh( struct libusb_device_handle* devh )
1014 {
1015     int r ;
1016
1017     r = libusb_control_transfer( devh , CTRLOUT, UBERTOOTHLAFH, 0, 0,
1018                                NULL, 0, 1000);
1019     if (r < 0) {
1020         if (r == LIBUSB_ERROR_PIPE) {
1021             fprintf(stderr , "control message unsupported\n");
1022         } else {
1023             show_libusb_error(r);
1024         }
1025         return r;
1026     }
1027     return 0;
1028 }
1029
1030 int cmd_hop( struct libusb_device_handle* devh )
1031 {
1032     uint8_t buffer [LIBUSB_CONTROL_SETUP_SIZE];
1033     struct libusb_transfer *xfer = libusb_alloc_transfer(0);
1034
1035     libusb_fill_control_setup( buffer , CTRLOUT, UBERTOOTHLHOP, 0, 0, 0);
1036     libusb_fill_control_transfer( xfer , devh , buffer , \

```

```

1037     callback , NULL, 1000);
1038 libusb_submit_transfer(xfer);
1039
1040 return 0;
1041 }
1042
1043 int32_t cmd_api_version( struct libusb_device_handle* devh) {
1044     unsigned char data[4];
1045     int r;
1046
1047     r = libusb_control_transfer(devh, CTRL_IN, \
1048         UBERTOOTH_GET_APIVERSION, 0, 0, data, 4, 3000);
1049     if (r < 0) {
1050         show_libusb_error(r);
1051         return r;
1052     }
1053     return data[0] | data[1] << 8 | data[2] << 16 | data[3] << 24;
1054 }
1055
1056 int ubertooth_cmd_sync( struct libusb_device_handle* devh,
1057                         uint8_t type,
1058                         uint8_t command,
1059                         uint8_t* data,
1060                         uint16_t size)
1061 {
1062     int r;
1063
1064     r = libusb_control_transfer(devh, type, command, 0, 0,
1065                                 data, size, 1000);
1066     if (r < 0) {
1067         if (r == LIBUSB_ERROR_PIPE) {
1068             fprintf(stderr, "control message unsupported\n");
1069         } else {
1070             show_libusb_error(r);
1071         }
1072         return r;
1073     }
1074     return 0;
1075 }
1076
1077 int ubertooth_cmd_async( struct libusb_device_handle* devh,
1078                         uint8_t type,
1079                         uint8_t command,
1080                         uint8_t* data,
1081                         uint16_t size)
1082 {
1083     int r = 0;
1084
1085     uint8_t buffer [LIBUSB_CONTROL_SETUP_SIZE + size];
1086     struct libusb_transfer* xfer = libusb_alloc_transfer(0);
1087
1088     libusb_fill_control_setup(buffer, type, command, 0, 0, size);
1089     if(size > 0)
1090         memcpy (&buffer [LIBUSB_CONTROL_SETUP_SIZE], data, size );
1091     libusb_fill_control_transfer(\
1092         xfer, devh, buffer, callback, NULL, 1000);
1093     xfer->status =
1094         LIBUSB_TRANSFER_FREE_BUFFER | LIBUSB_TRANSFER_FREE_TRANSFER;

```

```
1095     r = libusb_submit_transfer(xfer);  
1096  
1097     if (r < 0)  
1098         show_libusb_error(r);  
1099  
1100     return r;  
1101 }
```

1.7 host/libubertooth/src/ubertooth_interface.h

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2012 Dominic Spill
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #ifndef _UBERTOOTH_INTERFACE_H
22 #define _UBERTOOTH_INTERFACE_H
23
24 #include <stdint.h>
25
26 /* increment on every API change */
27 #define UBERTOOTH_API_VERSION 1
28
29 #define DMA_SIZE 50
30
31 #define NUM_BREDR_CHANNELS 79
32
33 /*
34  * CLK_TUNE_TIME is the duration in units of 100 ns that we reserve
35  * for tuning the radio while frequency hopping. We start the tuning
36  * process CLK_TUNE_TIME * 100 ns prior to the start of an upcoming
37  * time slot.
38 */
39 #define CLK_TUNE_TIME 2250
40 #define CLK_TUNE_OFFSET 200
41
42 enum ubertooth_usb_commands {
43     UBERTOOTH_PING = 0,
44     UBERTOOTH_RX_SYMBOLS = 1,
45     UBERTOOTH_TX_SYMBOLS = 2,
46     UBERTOOTH_GET_USRLED = 3,
47     UBERTOOTH_SET_USRLED = 4,
48     UBERTOOTH_GET_RXLED = 5,
49     UBERTOOTH_SET_RXLED = 6,
50     UBERTOOTH_GET_TXLED = 7,
51     UBERTOOTH_SET_TXLED = 8,
52     UBERTOOTH_GET_1V8 = 9,
53     UBERTOOTH_SET_1V8 = 10,
54     UBERTOOTH_GET_CHANNEL = 11,
55     UBERTOOTH_SET_CHANNEL = 12,
```

```

56 UBERTOOTH_RESET = 13,
57 UBERTOOTH_GET_SERIAL = 14,
58 UBERTOOTH_GET_PARTNUM = 15,
59 UBERTOOTH_GET_PAEN = 16,
60 UBERTOOTH_SET_PAEN = 17,
61 UBERTOOTH_GET_HGM = 18,
62 UBERTOOTH_SET_HGM = 19,
63 UBERTOOTH_TX_TEST = 20,
64 UBERTOOTH_STOP = 21,
65 UBERTOOTH_GET_MOD = 22,
66 UBERTOOTH_SET_MOD = 23,
67 UBERTOOTH_SET_ISP = 24,
68 UBERTOOTH_FLASH = 25,
69 BOOTLOADER_FLASH = 26,
70 UBERTOOTH_SPECAN = 27,
71 UBERTOOTH_GET_PALEVEL = 28,
72 UBERTOOTH_SET_PALEVEL = 29,
73 UBERTOOTH_REPEAT = 30,
74 UBERTOOTH_RANGE_TEST = 31,
75 UBERTOOTH_RANGE_CHECK = 32,
76 UBERTOOTH_GET_REV_NUM = 33,
77 UBERTOOTH_LED_SPECAN = 34,
78 UBERTOOTH_GET_BOARD_ID = 35,
79 UBERTOOTH_SET_SQUELCH = 36,
80 UBERTOOTH_GET_SQUELCH = 37,
81 UBERTOOTH_SET_BDADDR = 38,
82 UBERTOOTH_START_HOPPING = 39,
83 UBERTOOTH_SET_CLOCK = 40,
84 UBERTOOTH_GET_CLOCK = 41,
85 UBERTOOTH_BTLE_SNIFFING = 42,
86 UBERTOOTH_GET_ACCESS_ADDRESS = 43,
87 UBERTOOTH_SET_ACCESS_ADDRESS = 44,
88 UBERTOOTH_DO_SOMETHING = 45,
89 UBERTOOTH_DO_SOMETHING_REPLY = 46,
90 UBERTOOTH_GET_CRC_VERIFY = 47,
91 UBERTOOTH_SET_CRC_VERIFY = 48,
92 UBERTOOTH_POLL = 49,
93 UBERTOOTH_BTLE_PROMISC = 50,
94 UBERTOOTH_SET_AFHMAP = 51,
95 UBERTOOTH_CLEAR_AFHMAP = 52,
96 UBERTOOTH_READ_REGISTER = 53,
97 UBERTOOTH_BTLE_SLAVE = 54,
98 UBERTOOTH_GET_COMPILE_INFO = 55,
99 UBERTOOTH_BTLE_SET_TARGET = 56,
100 UBERTOOTH_BTLE_PHY = 57,
101 UBERTOOTH_WRITE_REGISTER = 58,
102 UBERTOOTH_JAM_MODE = 59,
103 UBERTOOTH_LEG = 60,
104 UBERTOOTH_AFH = 61,
105 UBERTOOTH_HOP = 62,
106 UBERTOOTH_TRIM_CLOCK = 63,
107 UBERTOOTH_GET_APL_VERSION = 64,
108 UBERTOOTH_WRITE_REGISTERS = 65,
109 UBERTOOTH_READ_ALL_REGISTERS = 66,
110 UBERTOOTH_RX_GENERIC = 67,
111 UBERTOOTH_TX_GENERIC_PACKET = 68,
112 UBERTOOTH_FIX_CLOCK_DRIFT = 69,
113 UBERTOOTH_BTLE_MULTI_SNIFFING= 70,

```

```

114  };
115
116 enum jam_modes {
117     JAM_NONE      = 0,
118     JAM_ONCE      = 1,
119     JAM_CONTINUOUS = 2,
120 };
121
122 enum modulations {
123     MOD_BT_BASIC_RATE = 0,
124     MOD_BT_LOW_ENERGY = 1,
125     MOD_80211_FHSS   = 2,
126     MOD_NONE         = 3
127 };
128
129 enum usb_pkt_types {
130     BR_PACKET    = 0,
131     LE_PACKET    = 1,
132     MESSAGE       = 2,
133     KEEP_ALIVE   = 3,
134     SPECAN        = 4,
135     LE_PROMISC   = 5,
136     EGO_PACKET   = 6,
137 };
138
139 enum hop_mode {
140     HOP_NONE      = 0,
141     HOP_SWEEP     = 1,
142     HOP_BLUETOOTH = 2,
143     HOP_BTLE      = 3,
144     HOP_BTLE_MULTI = 4,
145     HOP_DIRECT    = 5,
146     HOP_AFH       = 6,
147 };
148
149 enum usb_pkt_status {
150     DMA_OVERFLOW   = 0x01,
151     DMA_ERROR     = 0x02,
152     FIFO_OVERFLOW = 0x04,
153     CS_TRIGGER    = 0x08,
154     RSSI_TRIGGER  = 0x10,
155     DISCARD       = 0x20,
156 };
157
158 /*
159 * USB packet for Bluetooth RX (64 total bytes)
160 */
161 typedef struct {
162     u8      pkt_type;
163     u8      status;
164     u8      channel;
165     u8      clkn_high;
166     u32    clk100ns;
167     /* Max RSSI seen while collecting symbols in this packet */
168     char    rssi_max;
169     /* Min ... */
170     char    rssi_min;
171     /* Average ... */

```

```

172     char      rssi_avg;
173     /* Number of ... (0 means RSSI stats are invalid) */
174     u8       rssi_count;
175     u8       reserved[2];
176     u8       data[DMA_SIZE];
177 } usb_pkt_rx;
178
179 typedef struct {
180     u64      address;
181     u64      syncword;
182 } bdaddr;
183
184 typedef struct {
185     u8      valid;
186     u8      request_pa;
187     u8      request_num;
188     u8      reply_pa;
189     u8      reply_num;
190 } rangetest_result;
191
192 typedef struct {
193     u16      synch;
194     u16      syncl;
195     u16      channel;
196     u8       length;
197     u8       pa_level;
198     u8       data[DMA_SIZE];
199 } generic_tx_packet;
200
201 #endif /* _UBERTOOTH_INTERFACE_H */

```

1.8 host/libubertooth/src/ubertooth.c

```
1  /*
2   * Copyright 2016 Jose Gutierrez del Arroyo
3   * Copyright 2010, 2011 Michael Ossmann
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110-1301, USA.
19  */
20
21 #include <pthread.h>
22 #include <signal.h>
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <string.h>
26 #include <time.h>
27 #include <unistd.h>
28
29 #include "ubertooth.h"
30 #include "ubertooth_callback.h"
31 #include "ubertooth_control.h"
32 #include "ubertooth_interface.h"
33
34 #ifndef RELEASE
35 #define RELEASE "unknown"
36 #endif
37 #ifndef VERSION
38 #define VERSION "unknown"
39 #endif
40
41 uint32_t systime;
42 FILE* infile = NULL;
43 FILE* dumpfile = NULL;
44 int max_ac_errors = 2;
45
46 int do_exit = 1;
47 pthread_t poll_thread;
48
49 unsigned int packet_counter_max;
50
51 void print_version() {
52     printf("libubertooth %s (%s), libbtbb %s (%s)\n", VERSION, RELEASE,
53           btbb_get_version(), btbb_get_release());
54 }
55
56
```

```

57 ubertooth_t* cleanup_devh = NULL;
58 static void cleanup(int sig __attribute__((unused)))
59 {
60     if (cleanup_devh)
61         cleanup_devh->stop_ubertooth = 1;
62 }
63
64 static void cleanup_exit(int sig __attribute__((unused)))
65 {
66     if (cleanup_devh)
67         ubertooth_stop(cleanup_devh);
68     exit(0);
69 }
70
71 void register_cleanup_handler(ubertooth_t* ut, int do_exit) {
72     cleanup_devh = ut;
73
74     /* Clean up on ctrl-C. */
75     if (do_exit) {
76         signal(SIGINT, cleanup_exit);
77         signal(SIGQUIT, cleanup_exit);
78         signal(SIGTERM, cleanup_exit);
79     } else {
80         signal(SIGINT, cleanup);
81         signal(SIGQUIT, cleanup);
82         signal(SIGTERM, cleanup);
83     }
84 }
85
86 ubertooth_t* timeout_dev = NULL;
87 void stop_transfers(int sig __attribute__((unused))) {
88     if (timeout_dev)
89         timeout_dev->stop_ubertooth = 1;
90 }
91
92 void ubertooth_set_timeout(ubertooth_t* ut, int seconds) {
93     /* Upon SIGALRM, call stop_transfers() */
94     if (signal(SIGALRM, stop_transfers) == SIG_ERR) {
95         perror("Unable to catch SIGALRM");
96         exit(1);
97     }
98     timeout_dev = ut;
99     alarm(seconds);
100 }
101
102 static struct libusb_device_handle* \
103     find_ubertooth_device(int ubertooth_device)
104 {
105     struct libusb_context *ctx = NULL;
106     struct libusb_device **usb_list = NULL;
107     struct libusb_device_handle *devh = NULL;
108     struct libusb_device_descriptor desc;
109     int usb_devs, i, r, ret, ubertoohs = 0;
110     int ubertooth_devs[] = {0,0,0,0,0,0,0,0};
111
112     usb_devs = libusb_get_device_list(ctx, &usb_list);
113     for(i = 0 ; i < usb_devs ; ++i) {

```

```

114     r = libusb_get_device_descriptor(usb_list[i], &desc);
115     if(r < 0)
116         fprintf(stderr, "couldn't get usb descriptor for dev #%d!\n", i
117 );
117     if ((desc.idVendor == TC13_VENDORID && \
118             desc.idProduct == TC13_PRODUCTID)
119             || (desc.idVendor == U0_VENDORID && \
120                 desc.idProduct == U0_PRODUCTID)
121             || (desc.idVendor == U1_VENDORID && \
122                 desc.idProduct == U1_PRODUCTID))
123     {
124         ubertooth_devs[ubertoohths] = i;
125         ubertoohths++;
126     }
127 }
128 if(ubertoohths == 1) {
129     ret = libusb_open(usb_list[ubertooth_devs[0]], &devh);
130     if (ret)
131         show_libusb_error(ret);
132 }
133 else if (ubertoohths == 0)
134     return NULL;
135 else {
136     if (ubertooth_device < 0) {
137         fprintf(stderr, "multiple Ubertooth devices found!\
138             Use '-U' to specify device number\n");
139         uint8_t serial[17], r;
140         for(i = 0 ; i < ubertoohths ; ++i) {
141             libusb_get_device_descriptor(\
142                 usb_list[ubertooth_devs[i]], &desc);
143             ret = libusb_open(usb_list[ubertooth_devs[i]], &devh);
144             if (ret) {
145                 fprintf(stderr, " Device %d: ", i);
146                 show_libusb_error(ret);
147             }
148             else {
149                 r = cmd_get_serial(devh, serial);
150                 if(r==0) {
151                     fprintf(stderr, " Device %d: ", i);
152                     print_serial(serial, stderr);
153                 }
154                 libusb_close(devh);
155             }
156         }
157         devh = NULL;
158     } else {
159         ret =
160             libusb_open(usb_list[ubertooth_devs[ubertooth_device]], &
161             devh);
161         if (ret) {
162             show_libusb_error(ret);
163             devh = NULL;
164         }
165     }
166 }
```

```

167     return devh;
168 }
169
170 /*
171 * based on http://libusb.sourceforge.net/api-1.0/
172 group__asyncio.html#ga9fcb2aa23d342060ebda1d0cf7478856 */
173 static void rx_xfer_status(int status)
174 {
175     char *error_name = "";
176
177     switch (status) {
178         case LIBUSB_TRANSFER_ERROR:
179             error_name="Transfer error .";
180             break;
181         case LIBUSB_TRANSFER_TIMED_OUT:
182             error_name="Transfer timed out .";
183             break;
184         case LIBUSB_TRANSFER_CANCELLED:
185             error_name="Transfer cancelled .";
186             break;
187         case LIBUSB_TRANSFER_STALL:
188             error_name=
189                 "Halt condition detected , or control request not supported ."
190             ;
191             break;
192         case LIBUSB_TRANSFER_NO_DEVICE:
193             error_name="Device disconnected .";
194             break;
195         case LIBUSB_TRANSFER_OVERFLOW:
196             error_name="Device sent more data than requested .";
197             break;
198     }
199     fprintf(stderr , "rx_xfer status: %s (%d)\n" ,error_name ,status);
200 }
201
202 static void cb_xfer(struct libusb_transfer *xfer)
203 {
204     int r;
205     ubertooth_t* ut = (ubertooth_t*)xfer->user_data;
206
207     if (xfer->status != LIBUSB_TRANSFER_COMPLETED) {
208         if(xfer->status == LIBUSB_TRANSFER_TIMED_OUT) {
209             r = libusb_submit_transfer(ut->rx_xfer);
210             if (r < 0)
211                 fprintf(stderr , "Failed to submit USB transfer (%d)\n" , r);
212             return;
213         }
214         if(xfer->status != LIBUSB_TRANSFER_CANCELLED)
215             rx_xfer_status(xfer->status);
216         libusb_free_transfer(xfer);
217         ut->rx_xfer = NULL;
218         return;
219     }
220     if(ut->stop_ubertooh)
221         return;
222
223     fifo_inc_write_ptr(ut->fifo);

```

```

224     ut->rx_xfer->buffer = ( uint8_t *)fifo_get_write_element(ut->fifo) ;
225
226     r = libusb_submit_transfer(ut->rx_xfer) ;
227     if (r < 0)
228         fprintf(stderr, "Failed to submit USB transfer (%d)\n", r) ;
229 }
230
231 static void* poll_thread_main(void* arg __attribute__((unused)))
232 {
233     int r = 0;
234
235     while (!do_exit) {
236         struct timeval tv = { 1, 0 } ;
237         r = libusb_handle_events_timeout(NULL, &tv) ;
238         if (r < 0) {
239             do_exit = 1;
240             break;
241         }
242         usleep(1);
243     }
244
245     return NULL;
246 }
247
248 int ubertooth_bulk_thread_start()
249 {
250     do_exit = 0;
251
252     return pthread_create(&poll_thread, NULL, poll_thread_main, NULL);
253 }
254
255 void ubertooth_bulk_thread_stop()
256 {
257     do_exit = 1;
258
259     pthread_join(poll_thread, NULL);
260 }
261
262 int ubertooth_bulk_init(ubertooth_t* ut)
263 {
264     int r;
265
266     ut->rx_xfer = libusb_alloc_transfer(0);
267     libusb_fill_bulk_transfer(ut->rx_xfer, ut->devh, DATA_IN, \
268     (uint8_t *)fifo_get_write_element(ut->fifo), PKTLEN, \
269     cb_xfer, ut, TIMEOUT);
270
271     r = libusb_submit_transfer(ut->rx_xfer);
272     if (r < 0) {
273         fprintf(stderr, "rx_xfer submission: %d\n", r);
274         return -1;
275     }
276     return 0;
277 }
278
279 void ubertooth_bulk_wait(ubertooth_t* ut)
280 {
281     while (fifo_empty(ut->fifo) && !ut->stop_ubertooh)

```

```

282         usleep(1);
283     }
284
285     int ubertooth_bulk_receive(\n
286         ubertooth_t* ut, rx_callback cb, void* cb_args)
287     {
288         if (!fifo_empty(ut->fifo)) {
289             (*cb)(ut, cb_args);
290             if (ut->stop_ubertooth) {
291                 if (ut->rx_xfer)
292                     libusb_cancel_transfer(ut->rx_xfer);
293                 return 1;
294             }
295             fflush(stderr);
296             return 0;
297         } else {
298             usleep(1);
299             return -1;
300         }
301     }
302
303     static int stream_rx_usb(\n
304         ubertooth_t* ut, rx_callback cb, void* cb_args)
305     {
306         /* init USB transfer */
307         int r = ubertooth_bulk_init(ut);
308         if (r < 0)
309             return r;
310
311         r = ubertooth_bulk_thread_start();
312         if (r < 0)
313             return r;
314
315         /* tell ubertooth to send packets */
316         r = cmd_rx_syms(ut->devh);
317         if (r < 0)
318             return r;
319
320         /* receive and process each packet */
321         while (!ut->stop_ubertooth) {
322             ubertooth_bulk_wait(ut);
323             r = ubertooth_bulk_receive(ut, cb, cb_args);
324         }
325
326         ubertooth_bulk_thread_stop();
327
328         return 1;
329     }
330
331     /* file should be in full USB packet format (ubertooth-dump -f) */
332     int stream_rx_file(\n
333         ubertooth_t* ut, FILE* fp, rx_callback cb, void* cb_args)
334     {
335         uint8_t buf[PKTLEN];
336         size_t nitems;
337
338         while (1) {
339             uint32_t systime_be;

```

```

340     nitems = fread(&systime_be , sizeof(systime_be) , 1 , fp );
341     if (nitems != 1)
342         return 0;
343     systime = (time_t)be32toh(systime_be);
344
345     nitems = fread(buf , sizeof(buf[0]) , PKTLEN , fp );
346     if (nitems != PKTLEN)
347         return 0;
348     fifo_push(ut->fifo , (usb_pkt_rx*)buf);
349     (*cb)(ut , cb_args);
350 }
351 }
352
353 /* Receive and process packets. For now, returning from
354 * stream_rx_usb() means that UAP and clocks have been found, and that
355 * hopping should be started. A more flexible framework would be
356 * nice. */
357 void rx_live(ubertooth_t* ut , btbb_piconet* pn , int timeout)
358 {
359     int r = btbb_init(max_ac_errors);
360     if (r < 0)
361         return;
362
363     if (timeout)
364         ubertooth_set_timeout(ut , timeout);
365
366     if (pn != NULL && btbb_piconet_get_flag(pn , BTBB_CLK27_VALID))
367         cmd_set_clock(ut->devh , 0);
368     else {
369         stream_rx_usb(ut , cb_br_rx , pn);
370         /* Allow pending transfers to finish */
371         sleep(1);
372     }
373     /* Used when follow_pn is preset OR set by stream_rx_usb above
374      * i.e. This cannot be rolled in to the above if...else
375      */
376     if (pn != NULL && btbb_piconet_get_flag(pn , BTBB_CLK27_VALID)) {
377         ut->stop_ubertooth = 0;
378         /* cmd_stop(ut->devh); */
379         cmd_set_bdaddr(ut->devh , btbb_piconet_get_bdaddr(pn));
380         cmd_start_hopping(ut->devh , btbb_piconet_get_clk_offset(pn) , 0);
381         stream_rx_usb(ut , cb_br_rx , pn);
382     }
383 }
384
385 void rx_afh(ubertooth_t* ut , btbb_piconet* pn , int timeout)
386 {
387     int r = btbb_init(max_ac_errors);
388     if (r < 0)
389         return;
390
391     cmd_set_channel(ut->devh , 9999);
392
393     if (timeout) {
394         ubertooth_set_timeout(ut , timeout);
395
396         cmd_afh(ut->devh);

```

```

397     stream_rx_usb(ut, cb_afh_initial, pn);
398
399     cmd_stop(ut->devh);
400     ut->stop_ubertooth = 0;
401
402     btbb_print_afh_map(pn);
403 }
404
405 /*
406  * Monitor changes in AFH channel map
407  */
408 cmd_clear_afh_map(ut->devh);
409 cmd_afh(ut->devh);
410 stream_rx_usb(ut, cb_afh_monitor, pn);
411 }
412
413 void rx_afh_r(ubertooth_t* ut, btbb_piconet* pn, \
414                 int timeout __attribute__((unused)))
415 {
416     static uint32_t lasttime;
417
418     int r = btbb_init(max_ac_errors);
419     int i, j;
420     if (r < 0)
421         return;
422
423     cmd_set_channel(ut->devh, 9999);
424
425     cmd_afh(ut->devh);
426
427     /* init USB transfer */
428     r = ubertooth_bulk_init(ut);
429     if (r < 0)
430         return;
431
432     r = ubertooth_bulk_thread_start();
433     if (r < 0)
434         return;
435
436     /* tell ubertooth to send packets */
437     r = cmd_rx_syms(ut->devh);
438     if (r < 0)
439         return;
440
441     /* receive and process each packet */
442     while (!ut->stop_ubertooth) {
443         ubertooth_bulk_receive(ut, cb_afh_r, pn);
444         if (lasttime < time(NULL)) {
445             lasttime = time(NULL);
446             printf("%u ", (uint32_t)time(NULL));
447             /* btbb_print_afh_map(pn); */
448
449             uint8_t* afh_map = btbb_piconet_get_afh_map(pn);
450             for (i=0; i<10; i++)
451                 for (j=0; j<8; j++)
452                     if (afh_map[i] & (1<<j))
453                         printf("1");
454                     else

```

```

455         printf("0");
456         printf("\n");
457     }
458 }
459
460 ubertooth_bulk_thread_stop();
461 }
462
463 /* sniff one target LAP until the UAP is determined */
464 void rx_file(FILE* fp, btbb_piconet* pn)
465 {
466     int r = btbb_init(max_ac_errors);
467     if (r < 0)
468         return;
469
470     ubertooth_t* ut = ubertooth_init();
471     if (ut == NULL)
472         return;
473
474     stream_rx_file(ut, fp, cb_br_rx, pn);
475 }
476
477 void rx_btle_file(FILE* fp)
478 {
479     ubertooth_t* ut = ubertooth_init();
480     if (ut == NULL)
481         return;
482
483     stream_rx_file(ut, fp, cb_btle, NULL);
484 }
485
486 void ubertooth_unpack_symbols(const uint8_t* buf, char* unpacked)
487 {
488     int i, j;
489
490     for (i = 0; i < SYM_LEN; i++) {
491         /* output one byte for each received symbol (0x00 or 0x01) */
492         for (j = 0; j < 8; j++) {
493             unpacked[i * 8 + j] = ((buf[i] << j) & 0x80) >> 7;
494         }
495     }
496 }
497
498 static void cb_dump_bitstream(
499     ubertooth_t* ut, void* args __attribute__((unused)))
500 {
501     int i;
502     char nl = '\n';
503
504     usb_pkt_rx usb = fifo_pop(ut->fifo);
505     usb_pkt_rx* rx = &usb;
506     char bitstream[BANKLEN];
507     ubertooth_unpack_symbols((uint8_t*)rx->data, bitstream);
508
509     /* convert to ascii */
510     for (i = 0; i < BANKLEN; ++i)
511         bitstream[i] += 0x30;
512

```

```

513     fprintf(stderr, "rx block timestamp %u * 100 nanoseconds\n",
514             rx->clk100ns);
515     if (dumpfile == NULL) {
516         fwrite(bitstream, sizeof(uint8_t), BANKLEN, stdout);
517         fwrite(&nl, sizeof(uint8_t), 1, stdout);
518     } else {
519         fwrite(bitstream, sizeof(uint8_t), BANKLEN, dumpfile);
520         fwrite(&nl, sizeof(uint8_t), 1, dumpfile);
521     }
522 }
523
524 static void cb_dump_full(\n
525     ubertooth_t* ut, void* args __attribute__((unused)))
526 {
527     usb_pkt_rx usb = fifo_pop(ut->fifo);
528     usb_pkt_rx* rx = &usb;
529
530     fprintf(stderr, \
531             "rx block timestamp %u * 100 nanoseconds\n", rx->clk100ns);
532     uint32_t time_be = htobe32((uint32_t)time(NULL));
533     if (dumpfile == NULL) {
534         fwrite(&time_be, 1, sizeof(time_be), stdout);
535         fwrite((uint8_t*)rx, sizeof(uint8_t), PKTLEN, stdout);
536     } else {
537         fwrite(&time_be, 1, sizeof(time_be), dumpfile);
538         fwrite((uint8_t*)rx, sizeof(uint8_t), PKTLEN, dumpfile);
539         fflush(dumpfile);
540     }
541 }
542
543 /* dump received symbols to stdout */
544 void rx_dump(ubertooth_t* ut, int bitstream)
545 {
546     if (bitstream)
547         stream_rx_usb(ut, cb_dump_bitstream, NULL);
548     else
549         stream_rx_usb(ut, cb_dump_full, NULL);
550 }
551
552 void ubertooth_stop(ubertooth_t* ut)
553 {
554     /* make sure xfers are not active */
555     if (ut->rx_xfer != NULL)
556         libusb_cancel_transfer(ut->rx_xfer);
557     if (ut->devh != NULL) {
558         cmd_stop(ut->devh);
559         libusb_release_interface(ut->devh, 0);
560     }
561     libusb_close(ut->devh);
562     libusb_exit(NULL);
563
564     if (ut->h_pcap_bredr) {
565         btbb_pcap_close(ut->h_pcap_bredr);
566         ut->h_pcap_bredr = NULL;
567     }
568     if (ut->h_pcap_le) {

```

```

569     lell_pcap_close(ut->h_pcap_le);
570     ut->h_pcap_le = NULL;
571 }
572
573 if (ut->h_pcapng_bredr) {
574     btbb_pcapng_close(ut->h_pcapng_bredr);
575     ut->h_pcapng_bredr = NULL;
576 }
577 if (ut->h_pcapng_le) {
578     lell_pcapng_close(ut->h_pcapng_le);
579     ut->h_pcapng_le = NULL;
580 }
581 }
582
583 ubertooth_t* ubertooth_init()
584 {
585     ubertooth_t* ut = (ubertooth_t*)malloc(sizeof(ubertooth_t));
586     if(ut == NULL) {
587         fprintf(stderr, "Unable to allocate memory\n");
588         return NULL;
589     }
590
591     ut->fifo = fifo_init();
592     if(ut->fifo == NULL)
593         fprintf(stderr, "Unable to initialize ringbuffer\n");
594
595     ut->devh = NULL;
596     ut->rx_xfer = NULL;
597     ut->stop_ubertooth = 0;
598     ut->abs_start_ns = 0;
599     ut->start_clk100ns = 0;
600     ut->last_clk100ns = 0;
601     ut->clk100ns_upper = 0;
602
603     ut->h_pcap_bredr = NULL;
604     ut->h_pcap_le = NULL;
605     ut->h_pcapng_bredr = NULL;
606     ut->h_pcapng_le = NULL;
607
608     return ut;
609 }
610
611 int ubertooth_connect(ubertooth_t* ut, int ubertooth_device)
612 {
613     int r = libusb_init(NULL);
614     if (r < 0) {
615         fprintf(stderr, "libusb_init failed (got 1.0?)\n");
616         return -1;
617     }
618
619     ut->devh = find_ubertooth_device(ubertooth_device);
620     if (ut->devh == NULL) {
621         fprintf(stderr, "could not open Ubertooth device\n");
622         ubertooth_stop(ut);
623         return -1;
624     }
625
626     r = libusb_claim_interface(ut->devh, 0);

```

```

627     if (r < 0) {
628         fprintf(stderr, "usb_claim_interface error %d\n", r);
629         ubertooth_stop(ut);
630         return -1;
631     }
632     return 1;
633 }
634 }
635
636 ubertooth_t* ubertooth_start(int ubertooth_device)
637 {
638     ubertooth_t* ut = ubertooth_init();
639
640     int r = ubertooth_connect(ut, ubertooth_device);
641     if (r < 0)
642         return NULL;
643
644     return ut;
645 }
646
647 int ubertooth_check_api(ubertooth_t *ut) {
648     int r;
649
650     r = cmd_api_version(ut->devh);
651     if (r < 0) {
652         fprintf(stderr, "Ubertooth running very old firmware found.\n");
653         fprintf(stderr, "Please upgrade to latest released firmware.\n");
654         ubertooth_stop(ut);
655         return -1;
656     }
657     else if (r < UBERTOOTHAPIVERSION) {
658         fprintf(stderr, \
659             "Ubertooth API version %d found, libubertooth requires %d.\n",
660             r, UBERTOOTHAPIVERSION);
661         fprintf(stderr, "Please upgrade to latest released firmware.\n");
662         ubertooth_stop(ut);
663         return -1;
664     }
665     else if (r > UBERTOOTHAPIVERSION) {
666         fprintf(stderr, \
667             "Ubertooth API version %d found, newer than that \
668             supported by libubertooth (%d).\n",
669             r, UBERTOOTHAPIVERSION);
670         fprintf(stderr, "Things will still work, but you might want to \
671             update your host tools.\n");
672     }
673     return 0;
674 }
675
676 /* Return all connected Ubertooth devices */
677 int ubertooth_enumerate() {
678     return 0;
679 }

```

1.9 host/libubertooth/src/ubertooth.h

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2010 – 2013 Michael Ossmann, Dominic Spill,
4   * Will Code, Mike Ryan
5   *
6   * This program is free software; you can redistribute it and/or modify
7   * it under the terms of the GNU General Public License as published by
8   * the Free Software Foundation; either version 2, or (at your option)
9   * any later version.
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; see the file COPYING. If not, write to
18  * the Free Software Foundation, Inc., 51 Franklin Street,
19  * Boston, MA 02110–1301, USA.
20  */
21
22 #ifndef __UBERTOOTH_H__
23 #define __UBERTOOTH_H__
24
25 #include "ubertooth_control.h"
26 #include "ubertooth_fifo.h"
27 /*#include <btbb.h> */
28 #include <btbb.h>
29
30 /* specan output types
31  * see https://github.com/dkogan/feedgnuplot for plotter */
32 enum specan_modes {
33     SPECAN_STDOUT      = 0,
34     SPECAN_GNUPLOT_NORMAL = 1,
35     SPECAN_GNUPLOT_3D    = 2,
36     SPECAN_FILE         = 3
37 };
38
39 enum board_ids {
40     BOARD_ID_UBERTOOTH_ZERO = 0,
41     BOARD_ID_UBERTOOTH_ONE  = 1,
42     BOARD_ID_TC13BADGE     = 2
43 };
44
45 typedef struct {
46     /* Ringbuffers for USB and Bluetooth symbols */
47     fifo_t* fifo;
48
49     struct libusb_device_handle* devh;
50     struct libusb_transfer* rx_xfer;
51
52     uint8_t stop_ubertooth;
53     uint64_t abs_start_ns;
54     uint32_t start_clk100ns;
55     uint64_t last_clk100ns;
```

```

56     uint64_t clk100ns_upper;
57
58     btbb_pcap_handle* h_pcap_bredr;
59     lell_pcap_handle* h_pcap_le;
60     btbb_pcapng_handle* h_pcapng_bredr;
61     lell_pcapng_handle* h_pcapng_le;
62 } ubertooth_t;
63
64 typedef void (*rx_callback)(ubertooth_t* ut, void* args);
65
66 typedef struct {
67     unsigned allowed_access_address_errors;
68 } btle_options;
69
70 extern uint32_t systime;
71 extern FILE* infile;
72 extern FILE* dumpfile;
73 extern int max_ac_errors;
74
75 void print_version();
76 void register_cleanup_handler(ubertooth_t* ut, int do_exit);
77 ubertooth_t* ubertooth_init();
78 int ubertooth_enumerate();
79 int ubertooth_connect(ubertooth_t* ut, int ubertooth_device);
80 ubertooth_t* ubertooth_start(int ubertooth_device);
81 void ubertooth_stop(ubertooth_t* ut);
82 int ubertooth_check_api(ubertooth_t *ut);
83 void ubertooth_set_timeout(ubertooth_t* ut, int seconds);
84
85 int ubertooth_bulk_init(ubertooth_t* ut);
86 void ubertooth_bulk_wait(ubertooth_t* ut);
87 int ubertooth_bulk_receive(ubertooth_t* , rx_callback , void* );
88 int ubertooth_bulk_thread_start();
89 void ubertooth_bulk_thread_stop();
90
91 int stream_rx_file(ubertooth_t* ut,FILE* , rx_callback , void* );
92
93 void rx_live(ubertooth_t* ut, btbb_piconet* pn, int timeout);
94 void rx_file(FILE* fp, btbb_piconet* pn);
95 void rx_dump(ubertooth_t* ut, int full);
96 void rx_btle(ubertooth_t* ut);
97 void rx_btle_file(FILE* fp);
98 void rx_afh(ubertooth_t* ut, btbb_piconet* pn, int timeout);
99 void rx_afh_r(ubertooth_t* ut, btbb_piconet* pn, int timeout);
100
101 void ubertooth_unpack_symbols(const uint8_t* buf, char* unpacked);
102
103 #endif /* _UBERTOOTH_H_ */

```

1.10 host/ubertooth-tools/src/ubertooth-btle.c

```
1  /*
2   * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
3   * Copyright 2012–2016 Michael Ryan
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2, or (at your option)
8   * any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; see the file COPYING. If not, write to
17  * the Free Software Foundation, Inc., 51 Franklin Street,
18  * Boston, MA 02110–1301, USA.
19  */
20
21 #include "ubertooth.h"
22 #include "ubertooth_callback.h"
23 #include <ctype.h>
24 #include <err.h>
25 #include <getopt.h>
26 #include <string.h>
27 #include <unistd.h>
28 #include <stdlib.h>
29
30 int convert_mac_address(char *s, uint8_t *o) {
31     int i;
32
33     /* validate length */
34     if (strlen(s) != 6 * 2 + 5) {
35         printf("Error: MAC address is wrong length\n");
36         return 0;
37     }
38
39     /* validate hex chars and : separators */
40     for (i = 0; i < 6*3; i += 3) {
41         if (!isxdigit(s[i]) ||
42             !isxdigit(s[i+1])) {
43             printf("Error: MAC address contains invalid character(s)\n");
44             return 0;
45         }
46         if (i < 5*3 && s[i+2] != ':') {
47             printf("Error: MAC address contains invalid character(s)\n");
48             return 0;
49     }
50 }
51
52 /* sanity: checked; convert */
53 for (i = 0; i < 6; ++i) {
54     unsigned byte;
55     sscanf(&s[i*3], "%02x", &byte);
```

```

56     o[ i ] = byte;
57 }
58
59     return 1;
60 }
61
62 static void usage( void )
63 {
64     printf("ubertooth-btle - passive Bluetooth Low Energy monitoring\n");
65     printf("Usage:\n");
66     printf("\t-h this help\n");
67     printf("\n");
68     printf("\t\tMajor modes:\n");
69     printf("\t-f follow single LE connection\n");
70     printf("\t-m follow multiple LE connections\n");
71     printf("\t-p promiscuous: sniff active connections\n");
72     printf("\t-a[ address] get/set access address \
73             (example: -a8e89bed6)\n");
74     printf("\t-s<address> faux slave mode, using MAC addr \
75             (example: -s22:44:66:88:aa:cc)\n");
76     printf("\t-t<address> set connection following target \
77             (example: -t22:44:66:88:aa:cc)\n");
78     printf("\n");
79     printf("\t\tInterference (use with -f or -p):\n");
80     printf("\t-i interfere with one connection and return to idle\n");
81     printf("\t-I interfere continuously\n");
82     printf("\n");
83     printf("\t\tData source:\n");
84     printf("\t-U<0-7> set ubertooth device to use\n");
85     printf("\n");
86     printf("\t\tMisc:\n");
87     printf("\t-r<filename> capture packets to PCAPNG file\n");
88     printf("\t-q<filename> capture packets to PCAP file \
89             (DLT_BLUETOOTH_LL_WITH_PHDR)\n");
90     printf("\t-c<filename> capture packets to PCAP file (DLT_PPI)\n");
91     printf("\t-A<index> advertising channel index (default 37)\n");
92     printf("\t-v[01] verify CRC mode, get status or enable/disable\n");
93     printf("\t-x<n> allow n access address offenses (default 32)\n");
94
95     printf("\nIf an input file is not specified, an Ubertooth device \
96             is used for live capture.\n");
97     printf("In get/set mode no capture occurs.\n");
98 }
99
100 int main( int argc, char *argv[] )
101 {
102     int opt;
103     int do_follow, do_follow_multi, do_promisc;
104     int do_get_aa, do_set_aa;
105     int do_crc;
106     int do_adv_index;
107     int do_slave_mode;
108     int do_target;
109     enum jam_modes jam_mode = JAM_NONE;
110     char ubertooth_device = -1;
111     ubertooth_t* ut = ubertooth_init();

```

```

112
113     btle_options cb_opts = { .allowed_access_address_errors = 32 };
114
115     int r;
116     u32 access_address;
117     uint8_t mac_address[6] = { 0, };
118
119     do_follow = do_follow_multi = do_promisc = 0;
120     do_get_aa = do_set_aa = 0;
121     do_crc = -1; /* 0 and 1 mean set, 2 means get */
122     do_adv_index = 37;
123     do_slave_mode = do_target = 0;
124
125     while ((opt=getopt(argc, argv, "a:r:hfmpU:v::A:s:t:x:c:q:jJiI")) != \
126             EOF) {
127         switch(opt) {
128             /* User wants a specific access address */
129             case 'a':
130                 if (optarg == NULL) {
131                     do_get_aa = 1;
132                 } else {
133                     do_set_aa = 1;
134                     sscanf(optarg, "%08x", &access_address);
135                 }
136                 break;
137
138             /* User wants to follow connection */
139             case 'f':
140                 do_follow = 1;
141                 break;
142
143             /* User wants to follow connection */
144             case 'm':
145                 do_follow_multi = 1;
146                 break;
147
148             /* User wants to operate in promiscuous mode */
149             case 'p':
150                 do_promisc = 1;
151                 break;
152
153             /* User wants to provide a specific Ubertooth interface */
154             case 'U':
155                 ubertooth_device = atoi(optarg);
156                 break;
157
158             /* User wants to capture files to PCAPNG */
159             case 'r':
160                 if (!ut->h_pcapng_le) {
161                     if (lell_pcapng_create_file(\
162                         optarg, "Ubertooth", &ut->h_pcapng_le)) {
163                         err(1, "lell_pcapng_create_file: ");
164                     }
165                 }
166                 else {
167                     printf("Ignoring extra capture file: %s\n", optarg);
168                 }

```

```

169         break;
170
171     /* User wants to capture files to PCAP */
172     case 'q':
173         if (!ut->h_pcap_le) {
174             if (lell_pcap_create_file(optarg, &ut->h_pcap_le)) {
175                 err(1, "lell_pcap_create_file: ");
176             }
177         }
178         else {
179             printf("Ignoring extra capture file: %s\n", optarg);
180         }
181         break;
182
183     /* User wants to capture files to pcap ppi */
184     case 'c':
185         if (!ut->h_pcap_le) {
186             if (lell_pcap_ppi_create_file(optarg, 0, &ut->h_pcap_le)) {
187                 err(1, "lell_pcap_ppi_create_file: ");
188             }
189         }
190         else {
191             printf("Ignoring extra capture file: %s\n", optarg);
192         }
193         break;
194     /* User wants to ensure CRC is verified */
195     case 'v':
196         if (optarg)
197             do_crc = atoi(optarg) ? 1 : 0;
198         else
199             do_crc = 2; /* get */
200         break;
201
202     /* User wants to listen on a specific advertisement channel */
203     case 'A':
204         do_adv_index = atoi(optarg);
205         if (do_adv_index < 37 || do_adv_index > 39) {
206             printf("Error: advertising index must be 37, 38, or 39\n");
207             usage();
208             return 1;
209         }
210         break;
211
212     /* User wants to operate as a slave */
213     case 's':
214         do_slave_mode = 1;
215         r = convert_mac_address(optarg, mac_address);
216         if (!r) {
217             usage();
218             return 1;
219         }
220         break;
221
222     /* User wants to follow a specific target */
223     case 't':
224         do_target = 1;
225         r = convert_mac_address(optarg, mac_address);

```

```

226     if (!r) {
227         usage();
228         return 1;
229     }
230     break;
231
232     /* How many access address errors should we tolerate */
233     case 'x':
234         cb_opts.allowed_access_address_errors = (unsigned) atoi(optarg)
235         ;
236         if (cb_opts.allowed_access_address_errors > 32) {
237             printf("Error: can tolerate 0-32 access address bit errors\n"
238         );
239         usage();
240         return 1;
241     }
242     break;
243
244     /* Jamming stuff */
245     case 'i':
246     case 'j':
247         jam_mode = JAM_ONCE;
248         break;
249     case 'I':
250     case 'J':
251         jam_mode = JAM_CONTINUOUS;
252         break;
253     case 'h':
254     default:
255         usage();
256         return 1;
257     }
258
259     /* Connect to the Ubertooth One */
260     r = ubertooth_connect(ut, ubertooth_device);
261     if (r < 0) {
262         usage();
263         return 1;
264     }
265
266     /* Check the firmware and tell the user if they need to update */
267     r = ubertooth_check_api(ut);
268     if (r < 0)
269         return 1;
270
271     /* Clean up on exit. */
272     register_cleanup_handler(ut, 1);
273
274     if (do_target) {
275         r = cmd_btle_set_target(ut->devh, mac_address);
276         if (r == 0) {
277             int i;
278             printf("target set to: ");
279             for (i = 0; i < 5; ++i)
280                 printf("%02x:", mac_address[i]);
281             printf("%02x\n", mac_address[5]);
282         }
283     }

```

```

282     }
283
284     /* Check if the user asked for more than one option
285      * (FOLLOW, MULTI, PROMISC) */
286     if (!(do_follow ^ do_follow_multi ^ do_promisc)) {
287         printf("Error: must choose only one of -f, -m, -p.\n");
288         return 1;
289     }
290
291     if (do_follow || do_follow_multi || do_promisc) {
292         usb_pkt_rx rx;
293
294         r = cmd_set_jam_mode(ut->devh, jam_mode);
295         if (jam_mode != JAM_NONE && r != 0) {
296             printf("Jamming not supported\n");
297             return 1;
298         }
299         cmd_set_modulation(ut->devh, MOD_BT_LOW_ENERGY);
300
301         if (do_follow || do_follow_multi) {
302             u16 channel;
303             if (do_adv_index == 37)
304                 channel = 2402;
305             else if (do_adv_index == 38)
306                 channel = 2426;
307             else
308                 channel = 2480;
309             cmd_set_channel(ut->devh, channel);
310
311             printf("About to send the 'follow' command via USB\n");
312             /* Follow */
313             if (do_follow) cmd_bt_le_sniffing(ut->devh, 2);
314             else if (do_follow_multi) cmd_bt_le_multi_sniffing(ut->devh, 2);
315
316         } else {
317             /* Promiscuous */
318             cmd_bt_le_promisc(ut->devh);
319         }
320
321         while (1) {
322             int r = cmd_poll(ut->devh, &rx);
323             if (r < 0) {
324                 printf("USB error\n");
325                 break;
326             }
327             if (r == sizeof(usb_pkt_rx)) {
328                 fifo_push(ut->fifo, &rx);
329                 cb_bt_le(ut, &cb_opts);
330             }
331             usleep(500);
332         }
333         ubertooh_stop(ut);
334     }
335
336     if (do_get_aa) {
337         access_address = cmd_get_access_address(ut->devh);
338         printf("Access address: %08x\n", access_address);

```

```

339     return 0;
340 }
341
342 if (do_set_aa) {
343     cmd_set_access_address(ut->devh, access_address);
344     printf("access address set to: %08x\n", access_address);
345 }
346
347 if (do_crc >= 0) {
348     int r;
349     if (do_crc == 2) {
350         r = cmd_get_crc_verify(ut->devh);
351     } else {
352         cmd_set_crc_verify(ut->devh, do_crc);
353         r = do_crc;
354     }
355     printf("CRC: %sverify\n", r ? "" : "DO NOT ");
356 }
357
358 if (do_slave_mode) {
359     u16 channel;
360     if (do_adv_index == 37)
361         channel = 2402;
362     else if (do_adv_index == 38)
363         channel = 2426;
364     else
365         channel = 2480;
366     cmd_set_channel(ut->devh, channel);
367
368     cmd_bt_le_slave(ut->devh, mac_address);
369 }
370
371 if (!(do_follow || do_promisc || do_get_aa || do_set_aa ||
372       do_crc >= 0 || do_slave_mode || do_target || do_follow_multi
373     ))
374     usage();
375
376 return 0;
377 }
```

Bibliography

1. AAB Smart Tools, Airflow & Environmental Meter Product Specification, (aabsmart.com/uploads/3/5/3/7/3537963/abm-200-spec-v1.0-0415.pdf), 2016.
2. Abracon Corporation, Ceramic Surface Mount Processor Crystal, (www.abracon.com/Resonators/ABMM.pdf), 2012.
3. Adafruit Industries, Bluefruit LE Sniffer, (www.adafruit.com/product/2269), 2015.
4. Arm.com, Serial Wire Debug - ARM, (www.arm.com/products/system-ip/debug-trace/coresight-soc-components/serial-wire-debug.php), 2016.
5. C. Badenhop, J. Fuller, J. Hall, B. Ramsey and M. Rice, Evaluating ITU-T G.9959 Based Wireless Systems Used in Critical Infrastructure Assets, *International Conference on Critical Infrastructure Protection*, Springer International Publishing, 2015.
6. Bluetooth SIG, Specification of the Bluetooth System Core Version 4.0, (www.bluetooth.com/specifications/adopted-specifications), 2010.
7. Bluetooth SIG, Specification of the Bluetooth System Core Version 4.2, (www.bluetooth.com/specifications/adopted-specifications), 2014.
8. BlueZ Project, BlueZ: Official Linuz Bluetooth protocol stack, (bluez.org), 2016.
9. A. Chang, Your lock is about to get clever: 5 smart locks compared, *Wired*, (www.wired.com/2013/06/smart-locks/), June 19, 2013.

10. J. Cheng, Air Force gets serious about securing infrastructure, *Defense Systems*, (defensesystems.com/articles/2014/06/17/air-force-cyber-nexus-critical-infrastructure.aspx), June 17, 2014.
11. Cypress Semiconductor Corporation, BCM20702/CYW20702 Datasheet, (www.cypress.com/file/297961/download), 2016.
12. S. Das, K. Kant and N. Zhang, *Handbook on Securing Cyber-Physical Critical Infrastructure*, Elsevier, 2012.
13. C. Dubendorfer, B. Ramsey and M. Temple, An RF-DNA Verification Process for ZigBee Networks, *Proceedings of 2012 IEEE Military Communications Conference*, 2012.
14. K. Fawaz, K. Kim and K. Shin, Protecting Privacy of BLE Device Users, *Proceedings of the Twenty-Fifth USENIX Security Symposium*, pp. 1205–1221, 2016.
15. J. Fuller, B. Ramsey, J. Pecarina and M. Rice, Wireless Intrusion Detection of Covert Channel Attacks in ITU-T G.9959-Based Networks, *Proceedings of the Eleventh International Conference on Cyber Warfare and Security*, 2016.
16. A. Gogic, A. Mujcic, S. Ibric and N. Suljanovic, Performance Analysis of Bluetooth Low Energy Mesh Routing Algorithm in Case of Disaster Prediction, *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 10(6), pp. 983–989, 2016.
17. T. Goodspeed, S. Bratus, R. Melgares, R. Speers and W. Smith, Api-do: Tools for Exploring the Wireless Attack Surface in Smart Meters, *Proceedings of the Forty-Fifth Hawaii International Conference on System Sciences*, pp. 2133–2140, 2012.

18. Great Scott Gadgets, Software, firmware, and hardware designs for Ubertooth, (github.com/greatscottgadgets/ubertooth), 2010.
19. Z. Guo, I. Harris, L. Tsaur and X. Chen, An On-demand Scatternet Formation and Multi-hop Routing Protocol for BLE-based wireless Sensor Networks, *Proceedings of IEEE Wireless Communications and Networking Conference*, pp. 1590–1595, 2015.
20. J. Gutierrez del Arroyo and B. Ramsey, How Do I “BLE Hacking”? , presented at *DEF CON 24 Wireless Village*, (www.youtube.com/watch?v=oP6sx2c0brY), 2016.
21. J. Gutierrez del Arroyo, J. Bindewald and B. Ramsey, Securing Bluetooth Low Energy Enabled Industrial Monitors, *Proceedings of the Twelfth International Conference on Cyber Warfare and Security*, 2017.
22. J. Gutierrez del Arroyo, Issue #232, Sniffer does not return to correct channel in ubertooth-btle -f, (github.com/greatscottgadgets/ubertooth/issues/232), 2016.
23. J. Gutierrez del Arroyo, Issue #234, Potential bad memory access issue in BLE connection follower, (github.com/greatscottgadgets/ubertooth/issues/234), 2016.
24. J. Gutierrez del Arroyo, J. Bindewald, S. Graham and M. Rice, Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections, *Proceedings of the Eleventh Annual IFIP WG 11.10 Conference on Critical Infrastructure Protection*, Publication Pending.

25. J. Hall, B. Ramsey, M. Rice and T. Lacey, Z-Wave Network Reconnaissance and Transceiver Fingerprinting Using Software Defined Radios, *Proceedings of the Eleventh International Conference on Cyber Warfare and Security*, 2016.
26. R. Heydon, *Bluetooth Low Energy: The Developer's Handbook*, Pearson Education, Inc., Crawfordsville, Indiana, USA, 2013.
27. K. Higgins, Anatomy of a ‘Cyber-Physical’ Attack, *DARKReading*, (www.darkreading.com/vulnerabilities---threats/anatomy-of-a-cyber-physical-attack-/d/d-id/1318624), January 14, 2015.
28. J. Hughes, J. Yan and K. Soga, Development of Wireless Sensor Network Using Bluetooth Low Energy (BLE) for Construction Noise Monitoring, *International Journal on Smart Sensing and Intelligent Systems*, vol. 8(2), pp. 1379–1405, 2015.
29. S. Jourdois, BtleJuice: Bluetooth Smart (LE) Man-in-the-Middle Framework, (github.com/DigitalSecurity/btlejuice), 2016.
30. H. Kim, J. Lee and J. Jang, BLEmesh: A Wireless Mesh Network Protocol for Bluetooth Low Energy Devices, *Proceedings of the Third International Conference on Future Internet of Things and Cloud*, pp. 558–563, 2015.
31. R. Lamance, SecAF outlines top priorities during ’State of AF’ address, *Air Force News Service*, (www.af.mil/News/ArticleDisplay/tabid/223/Article/473409/secaf-outlines-top-priorities-during-state-of-af-address.aspx), February 24, 2014.
32. A. Mathur and T. Newe, Comparison and Overview of Wireless Sensor Network Systems for Medical Applications, *Proceedings of the Eighth International Conference on Sensing Technology*, pp. 272–277, 2014.

33. T. Middleman, The pros and cons of Bluetooth Low Energy, *Electronics Weekly*, (www.electronicsweekly.com/news/design/communications/pros-cons-bluetooth-low-energy-2014-10/), October 10, 2014.
34. S. Mistry, bleno, A Node.js module for implementing BLE (Bluetooth Low Energy) peripherals, (github.com/sandeepmistry/bleno), 2013.
35. Nordic Semiconductor, DFU Service Specification, (devzone.nordicsemi.com/documentation/nrf51/4.4.1/html/group__dfu__ble__service__spec.html), 2013.
36. Nordic Semiconductor, nRF Sniffer, (www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF-Sniffer), 2014.
37. Nordic Semiconductor, nRF51 Development Kit, (www.nordicsemi.com/eng/Products/nRF51-DK), 2014.
38. Nordic Semiconductor, nRF51822 Product Specification v3.1, (infocenter.nordicsemi.com/pdf/nRF51822_PS_v3.1.pdf), 2014.
39. Onset Computer Corporation, HOBO MX Temp/RH Data Logger (MX1101) Manual, (www.onsetcomp.com/node/24391), 2014.
40. Onset Computer Corporation, HOBOMobile User's Guide for iOS, (www.onsetcomp.com/manuals/17841-hobomobile-users-guide), 2014.
41. M. Ossman and D. Spill, Building an All-Channel Bluetooth Monitor, presented at *ShmooCon 5*, 2009.
42. M. Ossman and D. Spill, Project Ubertooth, An Open Source 2.4 GHz Wireless Development Platform Suitable for Bluetooth Experimentation, (ubertooth.sourceforge.net), 2014.

43. Pwnie Express, Blue Hydra, (github.com/pwnieexpress/blue_hydra), 2016.
44. A. Rose, J. Gutierrez del Arroyo and B. Ramsey, BlueFinder: A Range-finding Tool for Bluetooth Classic and Low Energy, *Proceedings of the Twelfth International Conference on Cyber Warfare and Security*, 2017.
45. A. Rose and B. Ramsey, Picking Bluetooth Low Energy Locks from a Quarter Mile Away, presented at *DEF CON 24* (media.defcon.org/DEF\%20CON\%2024/DEF\%20CON\%2024\%20presentations/), 2016.
46. M. Ryan, Bluetooth: With Low Energy comes Low Security, *Proceedings of the Seventh USENIX Conference on Offensive Technologies*, 2013.
47. Segger, J-Link Commander, (www.segger.com/j-link-commander.html), 2016.
48. J. Slawomir, gattacker: A Node.js Package for BLE Using Man-in-the-Middle and Other Attacks, (github.com/securing/gattacker), 2016.
49. B. Sun, L. Osborne, Y. Xiao and S. Guizani, Intrusion Detection Techniques in Mobile Ad Hoc and Wireless Sensor Networks, *IEEE Wireless Communications*, vol. 14(5), pp. 56–63, 2007.
50. P. Thulasiraman and K. White, Topology control of tactical wireless sensor networks using energy efficient zone routing, *Digital Communications and Networks*, vol. 2(1), pp. 1–14, 2016.
51. Texas Instruments, CC2400 Datasheet, (www.ti.com/lit/ds/symlink/cc2400.pdf), 2006.
52. Texas Instruments, CC2540 USB Evaluation Module Kit, (www.ti.com/tool/cc2540emk-usb), 2013.

53. Texas Instruments, SmartRF Protocol Packet Sniffer,
(www.ti.com/tool/packet-sniffer), 2014.
54. Transducers Direct, Data Sheet for CirrusSense TDWLB Series Wireless
Bluetooth Pressure Transducer,
(www.transducersdirect.com/wp-content/uploads/2013/09/TDWLB_9.16.pdf),
2013.
55. G. Valadon and P. Lalet, Scapy: the python-based interactive packet
manipulation program & library, (github.com/secdev/scapy), 2003.

REPORT DOCUMENTATION PAGE

*Form Approved
OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)			2. REPORT TYPE		3. DATES COVERED (From — To)	
23-03-2017			Master's Thesis		Oct 2015 — Mar 2017	
4. TITLE AND SUBTITLE			5a. CONTRACT NUMBER 5b. GRANT NUMBER 17G310 5c. PROGRAM ELEMENT NUMBER			
Enhancing Critical Infrastructure Security Using Bluetooth Low Energy Traffic Sniffers			5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER			
6. AUTHOR(S)			8. PERFORMING ORGANIZATION REPORT NUMBER Gutiérrez del Arroyo, José, A, Captain, USAF 10. SPONSOR/MONITOR'S ACRONYM(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765			
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			11. SPONSOR/MONITOR'S REPORT NUMBER(S) Department of Homeland Security ICS-CERT POC: Neil Hershfield, DHS ICS-CERT Technical Lead ATTN: NPPD/CSC/NCSD/US-CERT Mailstop: 0635, 245 Murray Lane, SW, Bldg 410, Washington, DC 20528 Email: ics-cert@dhs.gov Phone: 1-877-776-7585			
12. DISTRIBUTION / AVAILABILITY STATEMENT						
DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES						
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT						
<p>Bluetooth Low Energy (BLE) is a wireless communications protocol used in Critical Infrastructure (CI) applications. Based on recent research trends, it is likely that the next generation of wireless sensor networks, a CI application that the Department of Defense (DoD) regularly employs in surveillance and reconnaissance missions, will include BLE as an inter-sensor communications protocol. Thus, future U.S. military missions may be directly impacted by the security of BLE. One natural way to help protect BLE sensors is to use BLE traffic sniffers to detect attacks. The primary limitation with current sniffers is that they can only capture one connection at a time, making them impractical for applications employing multiple BLE devices. This work aims to overcome that limitation to help secure the types of BLE sensor networks employed by the DoD. First, this work identifies vulnerabilities and enumerates attack vectors against a BLE wireless industrial sensor, presenting a list of security "best practices" that vendors and end-users can follow and demonstrating how users can employ BLE sniffers to detect attacks. The work then introduces BLE-Multi, an enhancement to an open-source BLE sniffer that can simultaneously and reliably capture multiple connections. Finally, the work presents and executes a methodology to evaluate BLE sniffers. Under the evaluation conditions applied, BLE-Multi achieves simultaneous capture of multiple active connections, paving the way for automated defensive tools that can be used by the DoD and security community. The contributions within are published in one journal article and one conference paper and were presented at three conferences focused on wireless security and CI protection.</p>						
15. SUBJECT TERMS						
Bluetooth Low Energy, Critical Infrastructure, Industrial Monitors, Wireless Sensor Networks, BLE Security, BLE Attack Detection, BLE Traffic Sniffers						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE	U	207	Maj Jason M. Bindewald, AFIT/ENG	
U	U	U	U	207	19b. TELEPHONE NUMBER (include area code) (937) 255-3636, x4614; jason.bindewald@afit.edu	